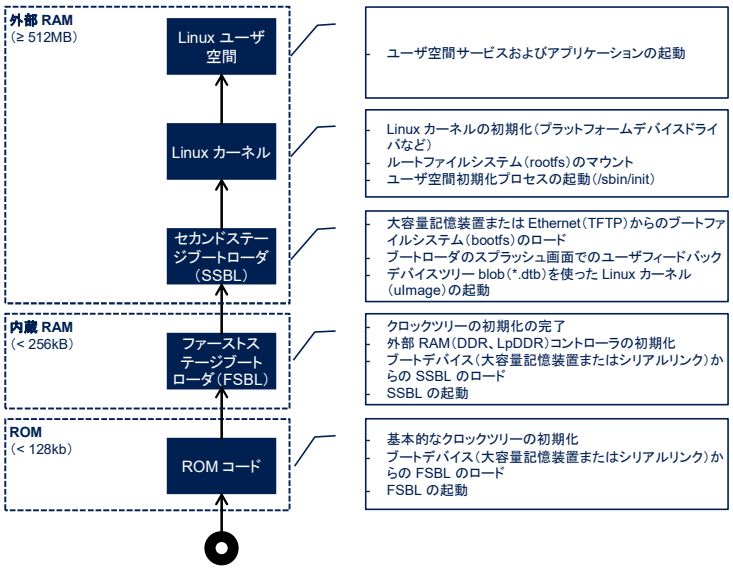
STM32MP1 プラットフォームのブート

プラットフォームのブート
1.0 版

life.augmented

こんにちは、STM32MP1 プラットフォームのブートを説明するプレゼンテーションへようこそ。

標準の Linux のブートチェーン



life.augmented

Linux プラットフォームのブートは、ロケットの打ち上げと非常に似ています。これは、最後の 1 つだけが最後に役立つ複数段階のプロセスであるからです。画面に表示されたブートチェーンは、市場で入手可能な他の MPU でも手順が似ているという意味で、まさに標準的です。

最初の段階は ROM コードです。このバイナリはマイクロプロセッサに組み込まれており、変更できないため、これはユーザの観点からはソフトウェアコンポーネントではありません。ROM コードは、ブート検出に関係するすべてのペリフェラルを有効にするために、最小限のクロックツリーを初期化します。これが完了すると、ファーストステージブートローダ(FSBL)がブートデバイスから内蔵 RAM にロードされ、実行されます。

FSBL は、ROM コードの直後に実行される最初の実際のソフトウェアコンポーネントです。多くのペリフェラル、その中でも、実行前にセカンドステージブートローダ(SSBL)をロードする外部 RAM を含むペリフェラルにアクセスするために、クロックツリーの初期化を完了します。

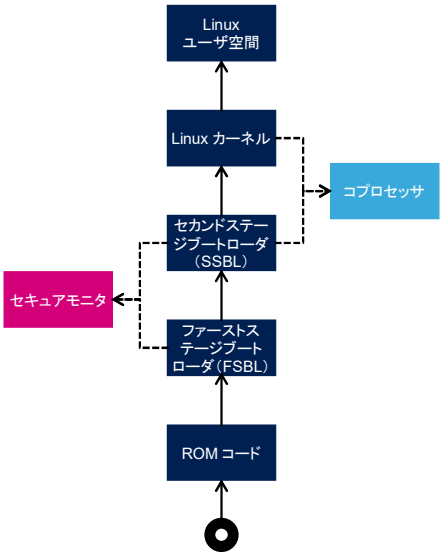
世界中で使用されている最も有名な SSBL は U-Boot です。その目的は、ROM コードでサポートされているブートデバイスのうちの特定のもの、または Ethernet などの他のブートデバイスから Linux カーネルを外部 RAM にロードすることです。SSBL には非常に大きな機能セットがあり、起動プロセス中にスプラッシュ画面と呼ばれる画像を表示するためによく使用されます。「ブートファイルシステム」すなわち「bootfs」には、U-Boot に必要な次のほとんどのバイナリが含まれています。スプラッシュ画面の画像、Linux カーネル、および Linux カーネルに与えられるすべての初期化データが入ったデバイスツリー blob などです。

U-Boot からの最後の動作は、Linux カーネルエントリポイントへのジャンプです … そして Linux が活動し始めます。カーネルは、すべてのデバイスドライバを初期化することから始め、次にすべてのユーザ空間アプリケーションとライブラリを含む「ルートファイルシステム」(「rootfs」)をマウントします。

ユーザ空間への切り替えは、Linux カーネルが「init」プロセスを作成するときに実現され、そのプロセスで rootfs に格納されているサービスとアプリケーションが起動されます。

この図は、ブートチェーンコンポーネントを組み込んだ連続するメモリの標準的なサイズを示しており、内部メモリでは数百キロバイト、外部メモリでは最大で数百メガバイトになります。

STM32MP1 ブートチェーン



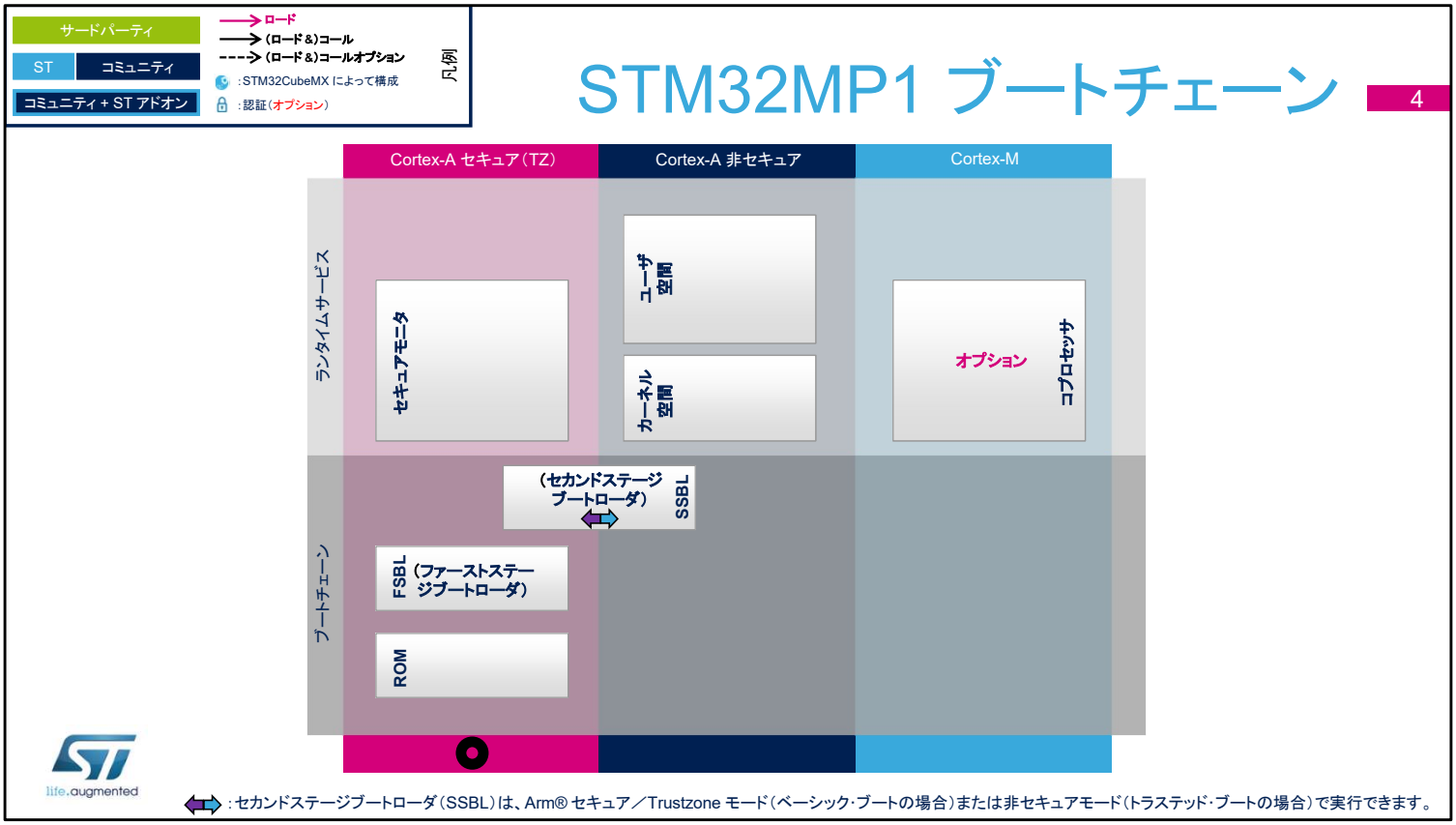
life.augmented

Linux の起動に加えて、STM32MP1 ブートチェーンは、プロセッサの他の 2 つの下記主要コンポーネントの起動にも関与します。

- Arm Cortex-A7 セキュアコンテキストによってサポートされるセキュアモニタ（TrustZone® とも呼ばれます）。このセキュアモニタは、ユーザ認証、鍵の保管、改ざん管理に使用できます。
- Arm Cortex-M4 コアで実行されるコプロセッサファームウェア。これは、リアルタイムサービスまたは低電力サービスの負荷軽減に使用できます。

図中の破線は以下のことを意味します。

- セキュアモニタは、ファーストステージブートローダまたはセカンドステージブートローダによって起動できます。
- コプロセッサは、セカンドステージブートローダ（「早期ブート」と呼ばれます）または Linux カーネルによって起動できます。



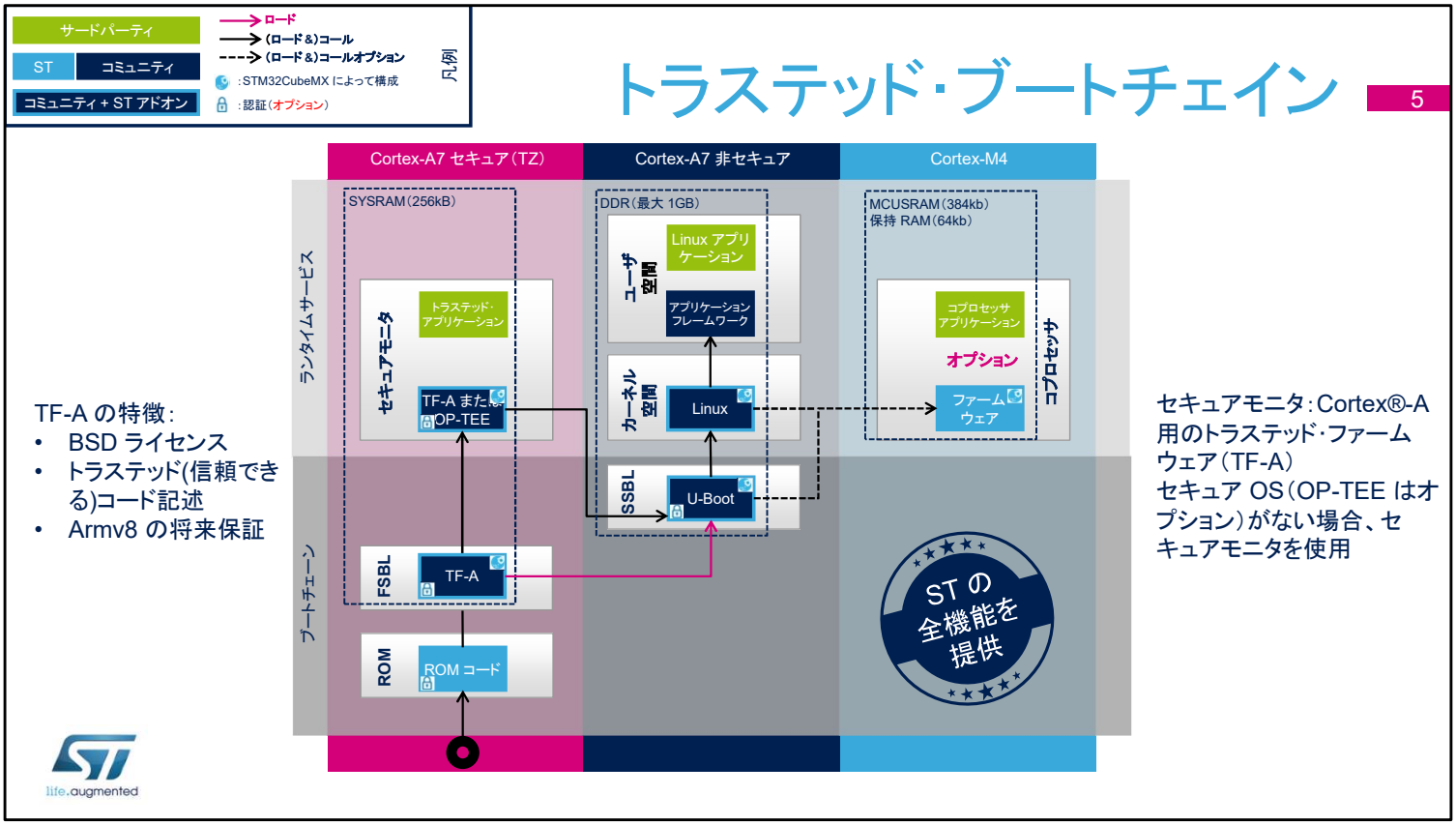
この図は、STM32MP1 プラットフォームでサポートされている以下の 3 つの「ハードウェア実行環境」を色付きの縦枠で紹介しています。

- ピンク色の Arm Cortex-A セキュア環境
- 紺色の Arm Cortex-A 非セキュア環境
- 水色の Arm Cortex-M 環境

灰色の横枠は、下部のブートチェーンと上部のランタイムサービスを示しています。

次に、前のスライドで紹介したいくつかのブートステージがこれらの縦枠と横枠に配置されています。ROM コード、ファーストステージブートローダ、セカンドステージブートローダ、Linux カーネル、ユーザ空間です。その上部の、左側にセキュアモニタ、右側にコプロセッサのファームウェアがあります。

ST マイクロエレクトロニクスは、STM32MP の FSBL と SSBL に 2 つの選択肢を提案しています。1 つの案は SSBL をセキュア側に配置し、もう 1 つの案は非セキュア側で実行することです。これが、SSBL ボックスの中の矢印の意味です。これらの 2 つのオプションについては、以降のスライドで詳しく説明します。



トラステッド・ブートチェーンは、ST マイクロエレクトロニクスが提供するデフォルトのソリューションであり、完全な機能セットを備えています。

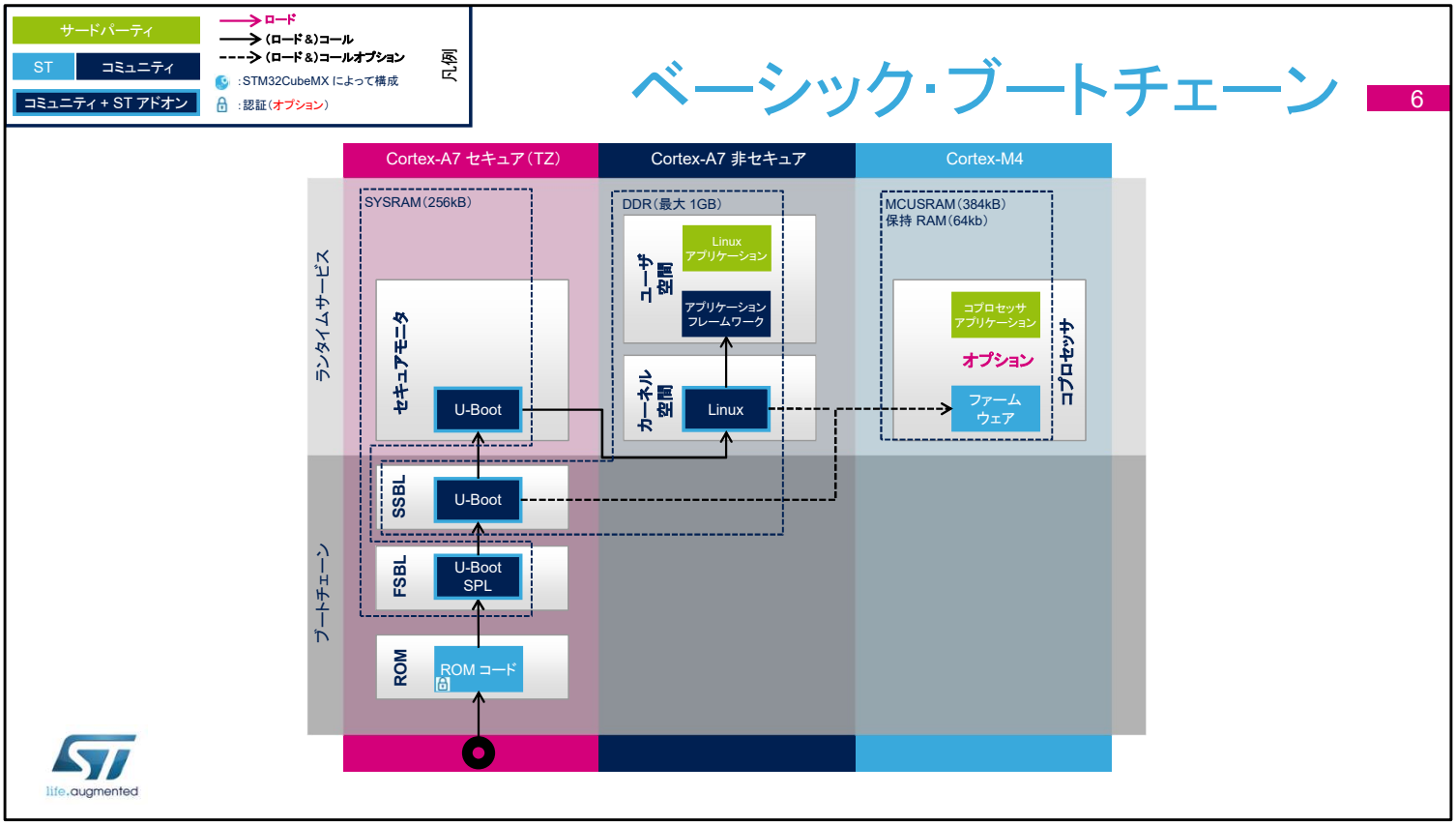
以下の理由で、FSBL として Arm の Cortex-A 用のトラステッド・ファームウェア(TF-A として知られている)を使用しています。

- このブートローダは BSD ライセンスの下で提供されます。これは、ブートチェーンの一部の詳細または実装を非表示にしたいお客様に好まれます。
- これは、信頼できることを目標にした Arm が開発したものであり、したがって、セキュリティの問題に敏感なお客様のすべての要件を満たすことができます。
- Armv8 アーキテクチャプラットフォームで広く使用されているため、将来性があります。

次に、トラステッド・ブートチェーンは SSBL として U-Boot を使用します。これは GPL v2 ライセンスで提供されています。

このブートチェーンでは認証はオプションであるため、セキュアブートオプションの有無にかかわらず、あらゆる STM32MP1 セキュリティバリアントで実行できます。

OP-TEE セキュア OS はオプションであり、STM32MP1 プラットフォームでトラステッド・アプリケーションを実行するためにのみ使用できます。それ以外の場合は、TF-A セキュアモニター(sp_min と呼ばれる)を使用して、プラットフォームをサポートするための最小限のセキュアサービスセットを実装します。



ベーシック·ブートチェーンは、単一のソースコードの U-Boot から FSBL と SSBL の両方を生成するためにも提案されています。U-Boot のセカンダリプログラムローダー(SPL)は、実際にファーストステージブートローダーとして使用できます。ST マイクロエレクトロニクスは、限られた数の機能を備えたベーシック·ブートチェーンをアップロードして、U-Boot コミュニティがそれを拡張できるようにしています。

ブートモード選択

BOOT pins	TAMP_REG[20] (Force Serial)	OTP WORD 3 Primary boot source	OTP WORD 3 Secondary boot source	Boot source #1	Boot source #2 if #1 fails	Boot source if #2 fails
b000	x (don't care)	x (don't care)	x (don't care)	Serial	-	-
b001	!= 0xFF	0 (virgin)	0 (virgin)	QSPI NOR	Serial	-
b010	!= 0xFF	0 (virgin)	0 (virgin)	eMMC	Serial	-
b011	!= 0xFF	0 (virgin)	0 (virgin)	FMC NAND	Serial	-
b100	x (don't care)	x (don't care)	x (don't care)	NoBoot	-	-
b101	!= 0xFF	0 (virgin)	0 (virgin)	SD-Card	Serial	-
b110	!= 0xFF	0 (virgin)	0 (virgin)	Serial	-	-
b111	!= 0xFF	0 (virgin)	0 (virgin)	QSPI NAND	Serial	-
!= b100	!= 0xFF	Primary[1]	0 (virgin)	Primary[1]	Serial	-
!= b100	!= 0xFF	0 (virgin)	Secondary[1]	Secondary[1]	Serial	-
!= b100	!= 0xFF	Primary[1]	Secondary[1]	Primary[1]	Secondary[1]	Serial
!= b100	0xFF	x (don't care)	x (don't care)	Serial	-	-

0	No secondary boot source is defined
1	FMC NAND

0	No primary boot source is defined
1	FMC NAND
2	QSPI NOR
3	eMMC
4	SD
5	QSPI NAND

[1]Primary and Secondary are fields of OTP WORD3.

life.augmented

参照: AN5031 - Getting started with STM32MP15 Series hardware development.
Wiki 記事: [STM32MP15_ROM_code_overview]

STM32MP1 ブートモードは、いくつかの下記入力の組み合わせによって定義されます。

- ST ボードでアクセス可能な 3 つのブートピン：それらの可能な値は、表の1列目に示されています。
- 次の列は TAMP バックアップレジスタ番号 20 に対応しており、U-Boot または Linux から 0xFF に設定されている場合、ユーザはシリアルブートを強制できます。
- 一度だけプログラム可能な WORD 3 には、プライマリブートソースとセカンダリブートソースが含まれており、それぞれ 3 列目と 4 列目に示されています。ブートソースの可能な値は、右側の表にリストされています：パラレル NAND Flash、QUADSPI NOR Flash、eMMC、SD カード、または QUADSPI NAND Flash です。

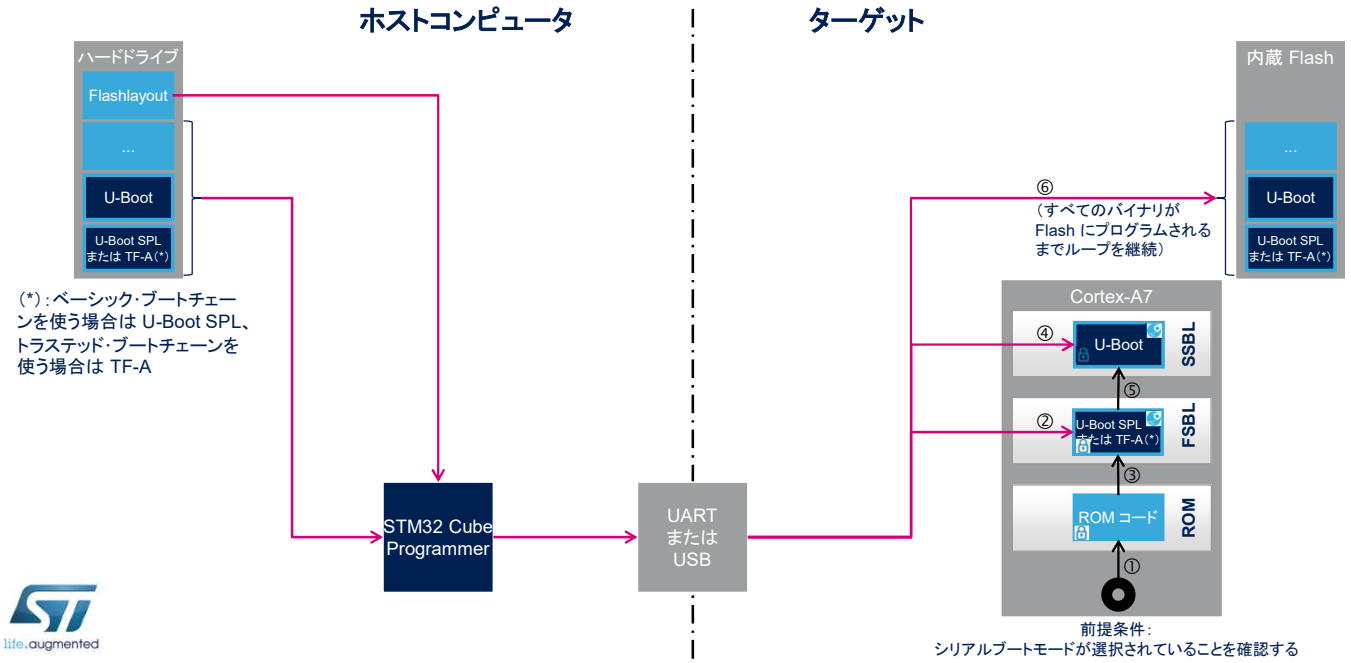
ブートピンには次の 2 つの特別なポジションがあります。

- すべてのピンがゼロの場合は強制的にシリアルモードでブートします。
- バイナリー値100 は、ブートなしモードに入ることを可能にし、Linux なしのファームウェア開発のために JTAG を介してコプロセッサーを取り扱うのに役立ちます。

緑色で囲まれている例を見てみましょう。常に SD カードでブートするようにボードを構成するには、これは右側の表のプライマリソース番号 4 であり、OTP WORD 3 にプライマリブートソースとして 4 を書き込みます。ROM コードが SD カードでのブートに成功しない場合、セカンダリブートソースが OTP で未設定状態であるため、シリアルブートにフォールバックします。

このスライドの表は、このスライドの下部にある ST Wiki の記事からコピーされているため、後で簡単に見つけることができます。

Flash のプログラミングのための STM32CubeProgrammer



このスライドでは、右側のボード組込み Flash を、左側のホストコンピュータ上に用意されたバイナリでプログラムするためにシリアルブートをどのように実施するかを説明します。

STM32CubeProgrammer は、Flash メモリをプログラムするために ST マイクロエレクトロニクスが提供するツールで、ホストコンピュータで動作するものです。

Flash メモリのプログラミングプロセスでは、UART または USB リンクを介して、ボードをホストコンピュータに接続する必要があります。次に、シリアルブートに対応するブートピンの組み合わせを選択し、ボードをリセットする必要があります。ここから、前述のようにブートチェーンを実行します。

1) ROM コードが起動し、選択されたブートモードが UART であれ USB であれ、シリアルであることが検出されます。ホストコンピュータから内蔵 RAM に利用可能なシリアルリンクを介して FSBL がダウンロードされ、実行されます。
2) FSBL は同様に、ホストコンピュータから SSBL を取得し、それを DDR にコピーして実行します。
3) SSBL はホストコンピュータに Flashlayout を要求します。Flashlayout には、期待される Flash メモリマッピングのテキストによる説明がパーティションごとに記載されています。
4) ブートチェーンは、Flashlayout からの指示に従って、Flash プログラミングプロセスが終了するまで SSBL のループにとどまります。

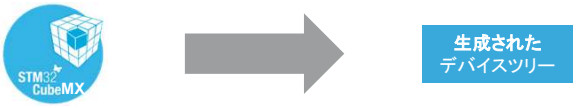
このシーケンスが完了すると、ユーザはブートピンを変更して、新しくプログラムされた Flash をブートデバイスとして選択し、そのボードをリセットしてブートできます。

デバイスツリーによる可変性の管理

- 以前の **Linux** カーネルはサポートされているボードのハードウェア記述を同じバイナリに埋め込んでいました。現在のカーネルでは、この情報を別のバイナリである**デバイスツリー blob**（dtb）に入れています。その結果、唯一のカーネルバイナリでさまざまなチップやボードをサポートできます。**U-Boot** も同じ解決策を採用。

- デバイスツリーの関連文書は、http://elinux.org/Device_Tree から入手可能。ThomasPetazzoni の Device Tree for Dummies を参照。

- Linux 開発者はデバイスツリーのソースファイル（dts）を手動で編集しますが、**STM32CubeMX** によって、この生成が可能になり、新しく始める方でも実践が容易になります。



life.augmented

次に、ブートチェーンの構成を見てみましょう。
どのソフトウェアも、適切に実行されるためには、実行されるプラットフォームのハードウェアの記述を取得する必要があります。これには、CPU の種類、メモリサイズ、ピン構成などが含まれます。

Linux カーネルはこのハードウェア記述をそのバイナリに直接埋め込んでいました。この歴史的な実装の結果、ハードウェアバリアントの管理は、新しいボードごとにカーネルをコンパイルするために必要なコンパイルスイッチに依存することがよくありました。

その後、デバイスツリーの概念が開発されました。ハードウェア構成をデバイスツリーのソースファイルに記述し、それをコンパイルしてデバイスツリー blob を得るというアイデアでした。この blob は、パラメータとして Linux カーネルに与えられ、カーネルは複数のプラットフォームで同じままにすることができます。たとえば、Arm-v7 アーキテクチャを実装するすべてのマイクロプロセッサは、multi_v7_config と呼ばれる唯一の Linux カーネル構成でサポートされます。

U-Boot は同じデバイスツリーの概念を採用しており、Arm は TF-A の同じトレンドに従っているため、ST マイクロエレクトロニクスは、DDR 構成を含むすべてのプラットフォーム構成データにデバイスツリーを広く使用しています。

Linux 開発者はデバイスツリーを手動で編集しています。一方、STM マイクロエレクトロニクスのお客様は、STM32CubeMX ツールを広く使用して、STM32 マイクロコントローラ用の STM32Cube ファームウェアを構成しています。そのため、DDR チューニング機能を備えたツールを拡張し、STM32MP1 マイクロプロセッサが内部ペリフェラルを構成するためのデバイスツリーを生成できるようにすることが決定されました。これにより、Linux 環境を検討している人にとって MCU の世界から MPU の世界へ移行することが容易になるはずです。

U-Boot バイナリと実行コンテキスト

- U-Boot ソースコードから、以下の **2 つのバイナリファイル**が生成

2 進数	実行コンテキスト	コメント
u-boot-spl.bin	FSBL	トラステッド･ブートチェーンは FSBL として TF-A を使用しているため、ベーシック･ブートチェーンにのみ適用できます。
u-boot.bin	SSBL pre-reloc	U-Boot は、FSBL によって読み込まれる場所である DDR の最初の位置から DDR の最後へ再配置されます。これにより、pre-reloc コンテキストが定義されます。
	SSBL	

- **デバイスツリー blob** は、各バイナリの最後に追加
  - u-boot-spl.bin デバイスツリーは U-Boot の **fdtgrep** ツールによって取得
    - fdtgrep は、「**u-boot,dm-spl**」または「**u-boot,dm-pre-reloc**」プロパティを持たないすべてのノードをフィルタリングして、「FSBL」コンテキストのために（狭い SYSRAM で実行されるので）可能な限りノードを軽減
  - u-boot.bin デバイスツリーはフィルタリングされませんが、以下の各コンテキストで使われ方が異なる
    - 「SSBL pre-reloc」は「**u-boot,dm-pre-reloc**」プロパティを持つノードのみを考慮
    - 「SSBL」コンテキストは完全なデバイスツリーを使用

life.augmented

U-Boot のコンパイルによって、以下の 2 つのバイナリが生成されます。

- u-boot-spl.bin（U-Boot セカンダリプログラムローダを表す）：これはベーシック･ブートチェーンで FSBL として使用されます。
- u-boot.bin、これは両方のブートチェーンで SSBL として使用され、ブート時に 2 回実行されます：
  - ❖ 最初の部分的な実行は、それ自体を DDR の末尾に再配置するために、FSBL によってコピーされた場所、つまり DDR ベースアドレスから開始されます。この実行コンテキストは「pre-reloc」と呼ばれます。
  - ❖ 2 番目の完全な実行は、再配置アドレスから始まります。

U-Boot で生成された各バイナリには、最後にデバイスツリー blob が追加されています。つまり、各バイナリには、実行可能コードと、実行可能コードによって探索されるデバイスツリー構成データが埋め込まれています。

U-Boot デバイスツリーには以下の 2 つの特別なプロパティが組み込まれています。

- 「U-Boot ドライバモデル-セカンダリプログラムローダ」を表す「u-boot,dm-spl」
- 「U-Boot ドライバモデル–再配置前」を表す「u-boot,dm-pre-reloc」

これらがどのように使用されるかを見てみましょう。

U-Boot SPL はサイズが制限された内部 RAM で実行されるため、FSBL コンテキストで役に立たないノードを削除するために、デバイスツリー blob をフィルタリングする必要があります。これは、「fdtgrep」U-Boot ツールを使用して行われ、「u-boot,dm-spl」または「u-boot,dm-pre-reloc」を持たないすべてのノードを dtb ファイルから削除します。

u-boot.bin にはこのメモリサイズの制約はありませんが、「pre-reloc」コンテキストであまりにも多くの初期化ステップを実行するには時間がかかります。そのため、U-Boot は「pre-reloc」コンテキストで実行されている間は「u-boot,dm-pre-reloc」タグ付きノードのみを考慮しますが、「pre-reloc」フェーズの後ではすべてのノードが考慮されます。

U-Boot の構成

- ビルド時の構成
  - ボードの定義
    - u-boot/include/configs/stm32mp*.h
      - メモリマッピング、ブートコマンド、有効にする機能（Kconfig にはない）
  - U-Boot の機能
    - u-boot/configs/stm32mp*_defconfig
      - ターゲットの選択、有効にする機能（配布、ブートディレイ、spl など）
      - menuconfig による変更
- 実行時の構成
  - デバイスツリー（次のスライドを参照）
    - defconfig または make オプションの DEVICE_TREE で選択
    - U-Boot バイナリの後に追加
    - U-Boot は、デバイスツリーの利用者（そのニーズ）及び、供給者（Linux カーネル）向け
  - DISTRO（doc/README.distro を参照）
    - defconfig による有効化
    - 原則は、起動メニューを定義するブートファイルシステムに extlinux.conf ファイルを埋め込む
    - ブートした配布モードごとに Linux カーネル／デバイスツリー／ブートコマンドの追加を選択可

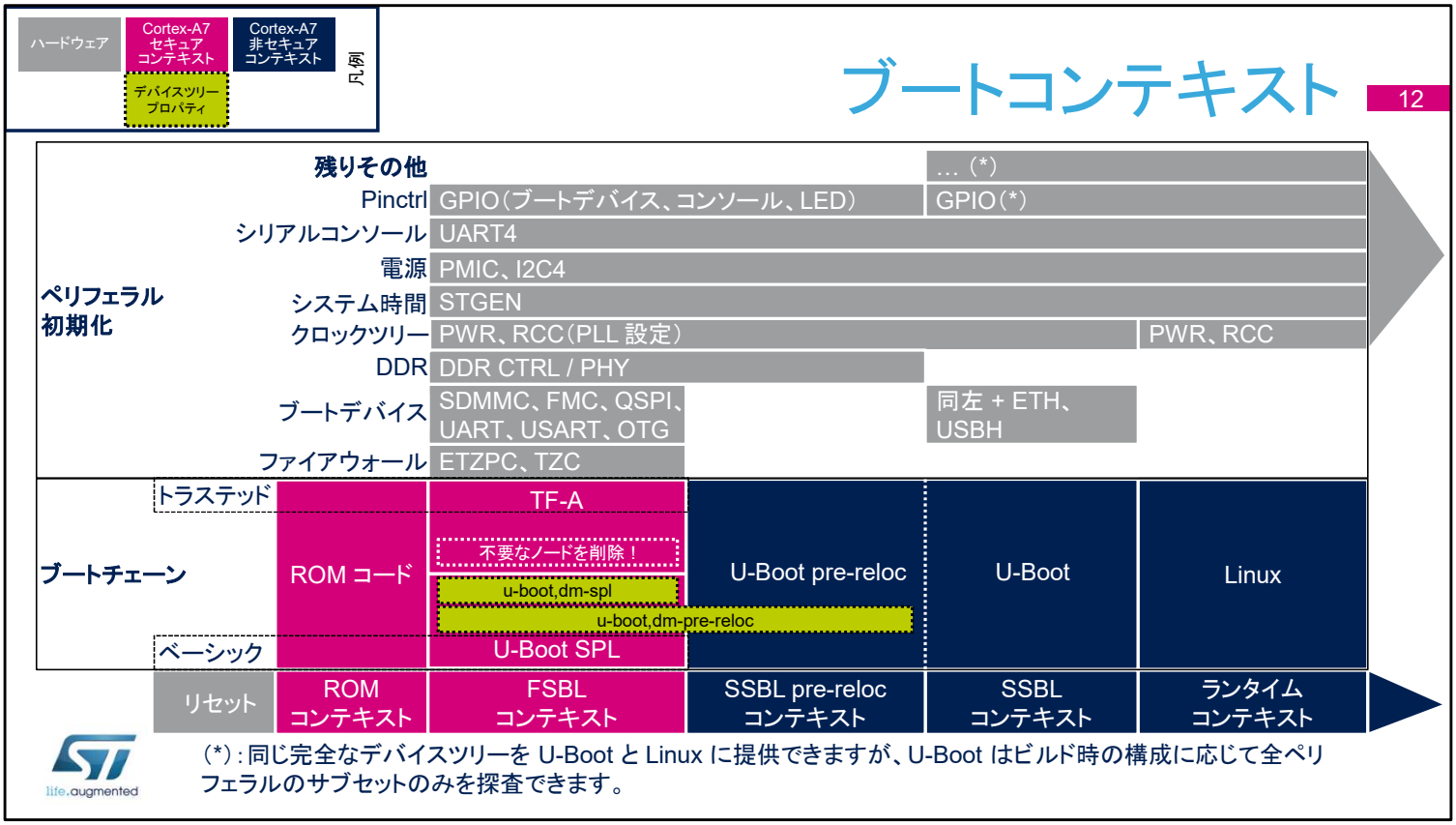
次に、U-Boot の動作を調整して、すべてのニーズに合わせる方法を見てみましょう。

ビルド時に、次のように U-Boot をカスタマイズできます。

- 専用メモリマッピング、ブートコマンドなどによってボード構成ファイルを定義します。
- menuconfig コマンドを使用してターゲットを選択し、次に説明する「distro」などの、埋め込む機能を決定します。

実行時に、下記により U-Boot の動作を変更できます。

- デバイスツリー（以前のスライドを参照）
- いくつかのブート構成のロードを可能にし、シリアルコンソールのブート時に選択メニュー（ユーザは使用したいものを選択できる）を表示する「distro」機能。たとえば、SD カード上に存在する Linux カーネルをブートする構成と、ネットワークから、すなわち、それがコンパイルされた開発者ホストコンピュータから直接 Linux カーネルをロードできるもう 1 つの構成を想像してみてください。



このスライドは、ブートチェーンの実行中に初期化されるペリフェラルのグループを示しています。

そこには大量の情報が含まれているので、まず左上隅にある凡例の説明を始めましょう。
- 灰色はハードウェアブロックに使われています。
- ピンク色は、Cortex-A7 セキュアコンテキストを強調するために使用されています。
- 紺色は、Cortex-A7 非セキュアコンテキストを強調するために使用されています。
- 破線は、デバイスツリーのカスタマイズを識別するために使用され、特に U-Boot の「u-boot,dm-spl」および「u-boot,dm-pre-reloc」プロパティに関して使用されています。

このスライドを下から読み始めて、上のフレームに向かって順に見ていきましょう。

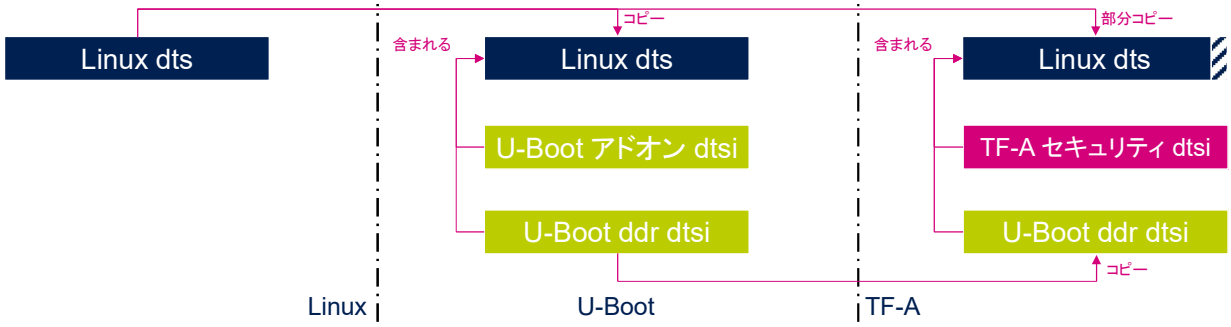
下の軸は、ブートチェーンの実行中に出会うさまざまな実行コンテキストを示しています。リセットから開始すると、ROM コンテキスト、FSBL、SSBL pre-reloc、および再配置後の SSBL があり、最終的に Linux が動作しているランタイムコンテキストにつながります。

そのすぐ上に、「u-boot,dm-spl」および「u-boot,dm-pre-reloc」プロパティの適用範囲とともに、これらのコンテキストの上にトラステッド·ブートチェーンおよびベーシック·ブートチェーンのコンポーネントのマッピングがあります。TF-A には U-Boot fdtgrep ツールに相当するものがないため、デバイスツリーのサイズを最適化する唯一の方法は、dts ファイルから不要なノードを直接削除することです。

このスライドの上部は、ROM コードコンテキスト以降の、ペリフェラルの初期化順序を示しています。
- ファイアウォールは、どのコンテキストでどのペリフェラルを使用できるかを定義することで構成されているため、非常に早期に FSBL によってセットアップされます。
- FSBL は ROM コードと同じブートデバイスのセットをサポートしますが、SSBL はこのリストに USB ホストと Ethernet を追加します。
- DDR コントローラの初期化は FSBL によって適用され、再配置の実行に使用される DDR サイズが含まれているため、pre-reloc コンテキストで参照できるようになっています。
- クロックツリーは FSBL によって適用される主な構成の 1 つです。最終的には、後で実行時に変更される可能性があります。
- Cortex-A7 汎用タイマのシステム時間は、STGEN によって提供され、FSBL で初期化されます。
- 電源は、ST ボード上の I2C4 を介して制御される外部 PMIC によって提供されます。したがって、どちらも最初に FSBL で初期化され、実行時にアプリケーションのニーズに沿ってすべて更新されます。
- UART4 は、ST ボード上のすべてのコンテキストからシリアルコンソールとして使用されます。
- GPIO は、コンテキストごとに段階的に初期化されます。
- 他のすべてのペリフェラルは、必要に応じて U-Boot または Linux によって初期化されます。

Linux、U-Boot、TF-A 用のデバイスツリー

13



- Linux では、すべての STM32MP1 プラットフォームが、1 式のデバイスツリーソースファイル（dts）を介してサポート
- U-Boot では、Linux dts ファイルがコピーされ、U-Boot アドオンプロパティと DDR 構成でオーバーロード
- TF-A では、Linux ファイルは部分的にコピーされ、DDR 構成（U-Boot からコピー）とセキュリティ構成（ファイアウォール）で完成

life.augmented

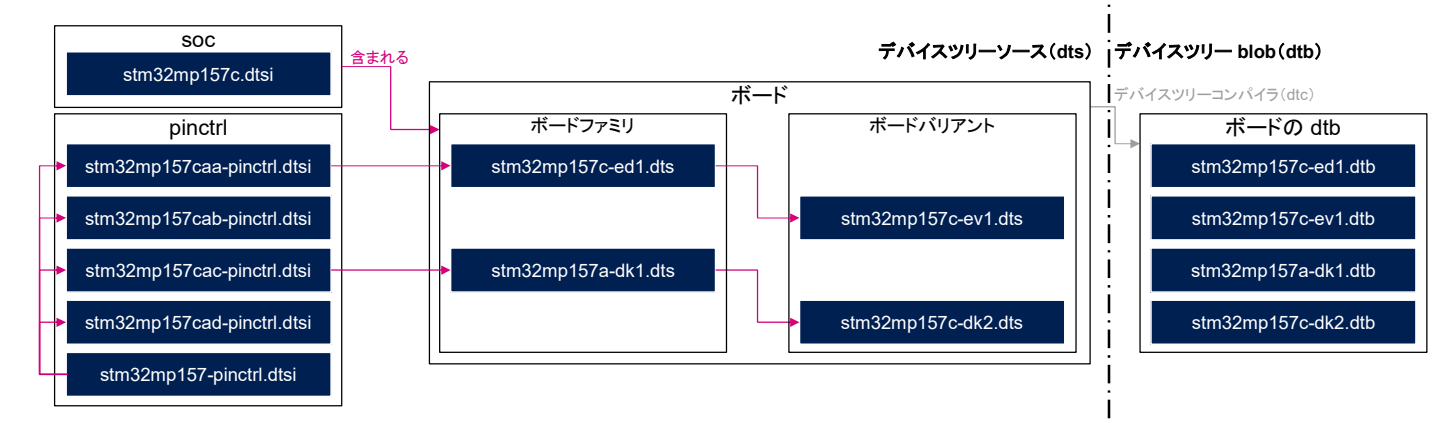
ブートプロセスは、以前に初期化されたペリフェラルのリストに新しいペリフェラルが追加される段階的なアプローチです。
しかし、対応するデバイスツリーファイルの開発フローは、以下のように逆の順序で行われます。

- Linux デバイスツリーのソースファイルを作成します。
- Linux ファイルが U-Boot にコピーされ、その後、下記によって完成されます。
  - ❖ 「u-boot,dm-spl」と「u-boot,dm-pre-reloc」を含む U-Boot アドオン
  - ❖ DDR 設定
- Linux ファイルが TF-A に部分的にコピーされ、その後、下記によって完成されます。
  - ❖ セキュリティパラメータ
  - ❖ DDR 設定

- ed：EVALドーターボード（STM32MP1 を搭載）
- ev：EVALボード（ed がプラグオンされたマザーボード）
- dk：ディスカバリキット

アップロードされたデバイスツリー

Linux



life.augmented

Linux デバイスツリーを詳しく見ていきましょう。これは次のようにいくつかのファイルに分割されています。
「soc」ファイルは STM32MP15 ペリフェラル定義に対応し、そのほとんどはこのファイルで「無効」になっています。
「pinctrl」は、STM32MP15 パッケージごとの GPIO バンクと、ボード上への取付け時に各パッケージで使用されるピン構成を定義します。
「ボード」ファイルは、次の 2 つのレベルに分割されています。
ボードファミリに共通するすべてを要因別に分解する「ボード」ファミリ
ファミリ全体にわたる違いを管理できる「ボードバリアント」

最後に、パラメーターとして Linux カーネルへ与えられるデバイスツリー blob を得るために、各ボードをデバイスツリーコンパイラ dtc でコンパイルすることができます。

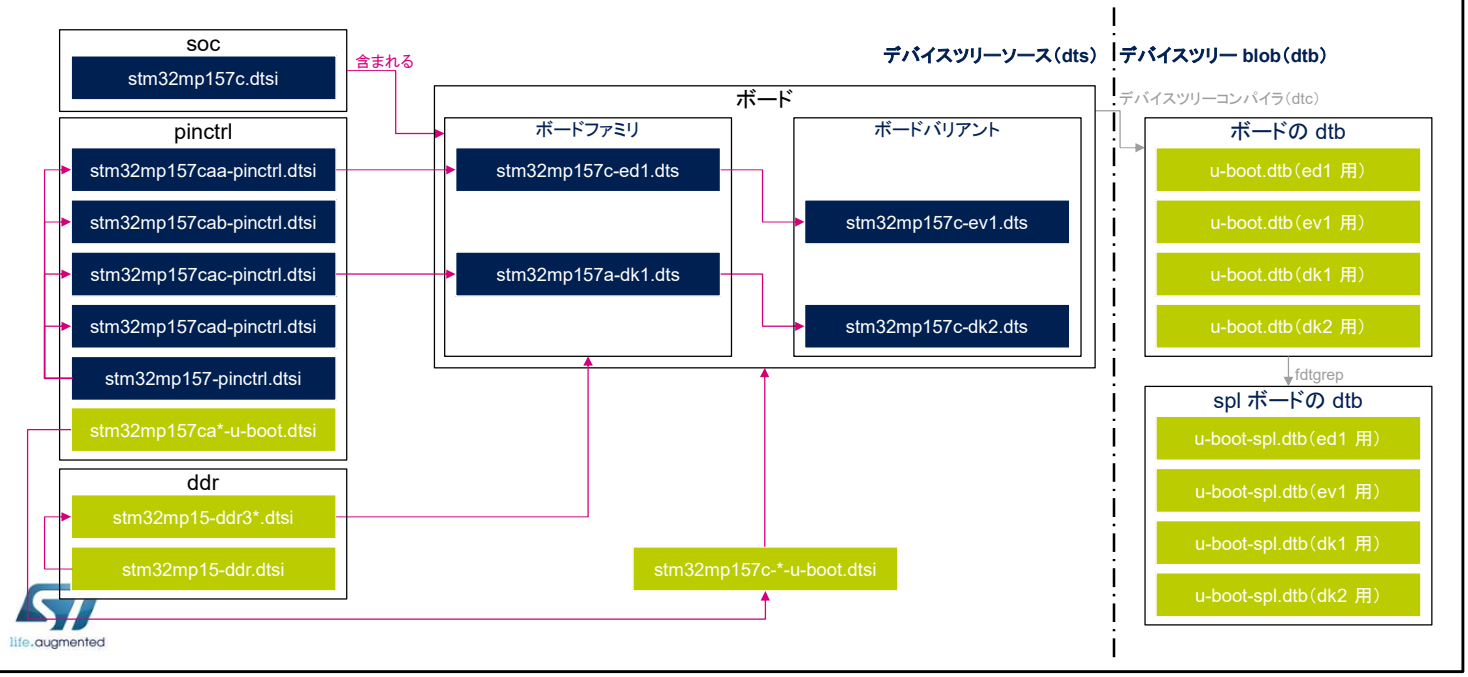
この図は、EVALボード「ev1」とディスカバリキット「dk2」のケースを対象としています。これらのボードは、EVALドーターボード「ed1」とディスカバリキット「dk1」のバリアントです。

ST マイクロエレクトロニクスは、オープンソースソフトウェアの使用とコミュニティへのアップロードを最大限に推進しています。したがって、これらすべてのデバイスツリーファイルは Linux リポジトリにアップロードされています。これは、U-Boot と TF-A に関する次の 2 つのスライドにも当てはまります。

- ed：EVALドーターボード（STM32MP1 を搭載）
- ev：EVALボード（ed がプラグオンされたマザーボード）
- dk：ディスカバリキット

アップロードされたデバイスツリー

U-Boot：Linux ファイルのコピーをオーバロード



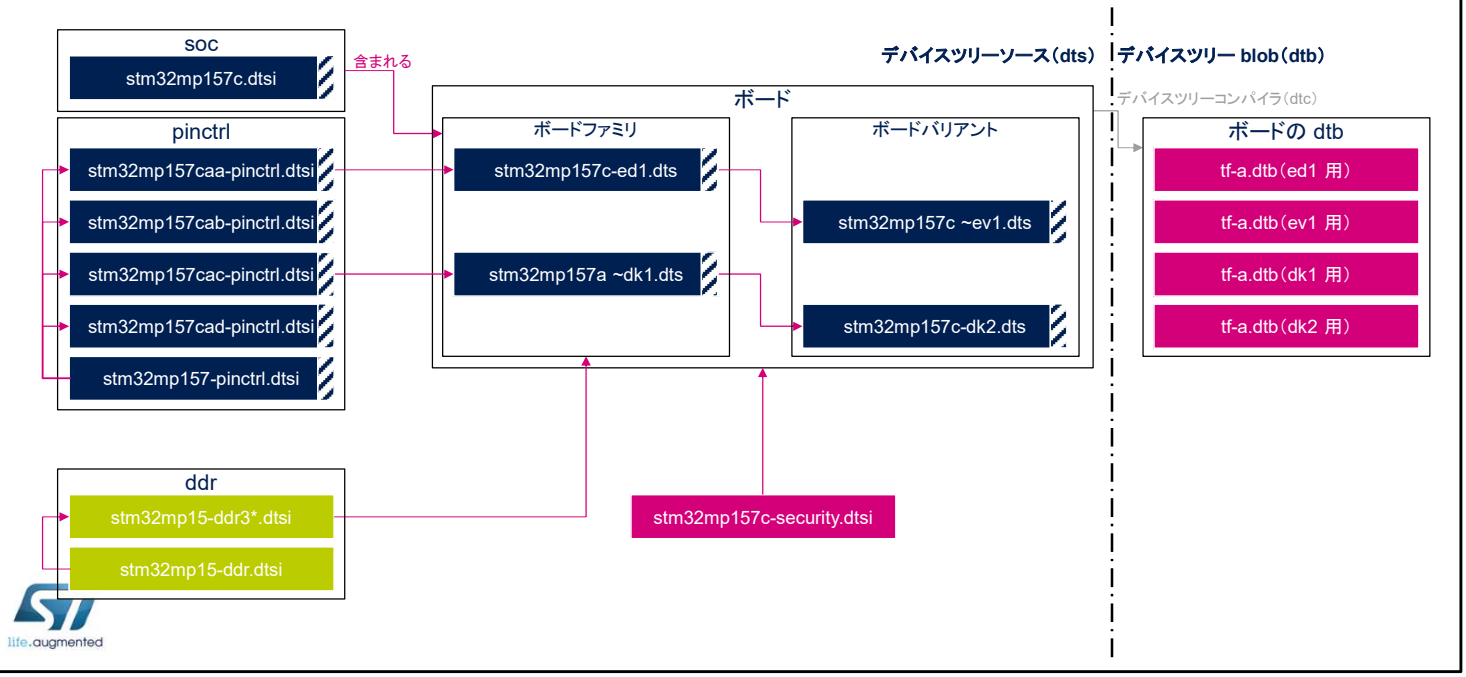
Linux デバイスツリーファイルのセットから始めて、この図は、主に U-Boot の「u-boot,dm-spl」および「u-boot,dm-pre- reloc」プロパティによっていくつかのオーバーレイを行うために U-Boot で追加されるファイルだけでなく、DDR コントローラの初期化用に追加されるファイルも示しています。

右側のビルドプロセスでは、各ボードの u-boot.dtb ファイルと u-boot-spl.dtb ファイルが生成されます。

- ed：EVALドーターボード（STM32MP1 を搭載）
- ev：EVALボード（ed がプラグオンされたマザーボード）
- dk：ディスカバリキット

アップロードされたデバイスツリー

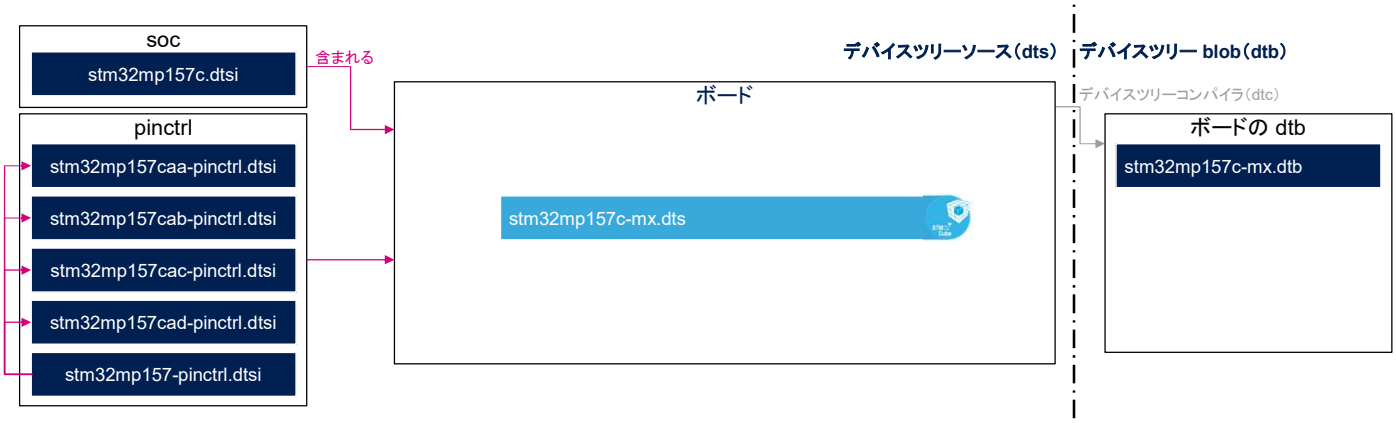
TF-A：Linux ファイルのサブセットをオーバーロードと、U-Boot からの DDR 構成を使用



Linux デバイスツリーファイルの内容の部分セットから始めて、この図は、TF-A でどのファイルが追加されているかを示し、U-Boot からの DDR 設定が再使用され、セキュリティペリフェラル構成が追加されていることを示しています。

STM32CubeMX による生成

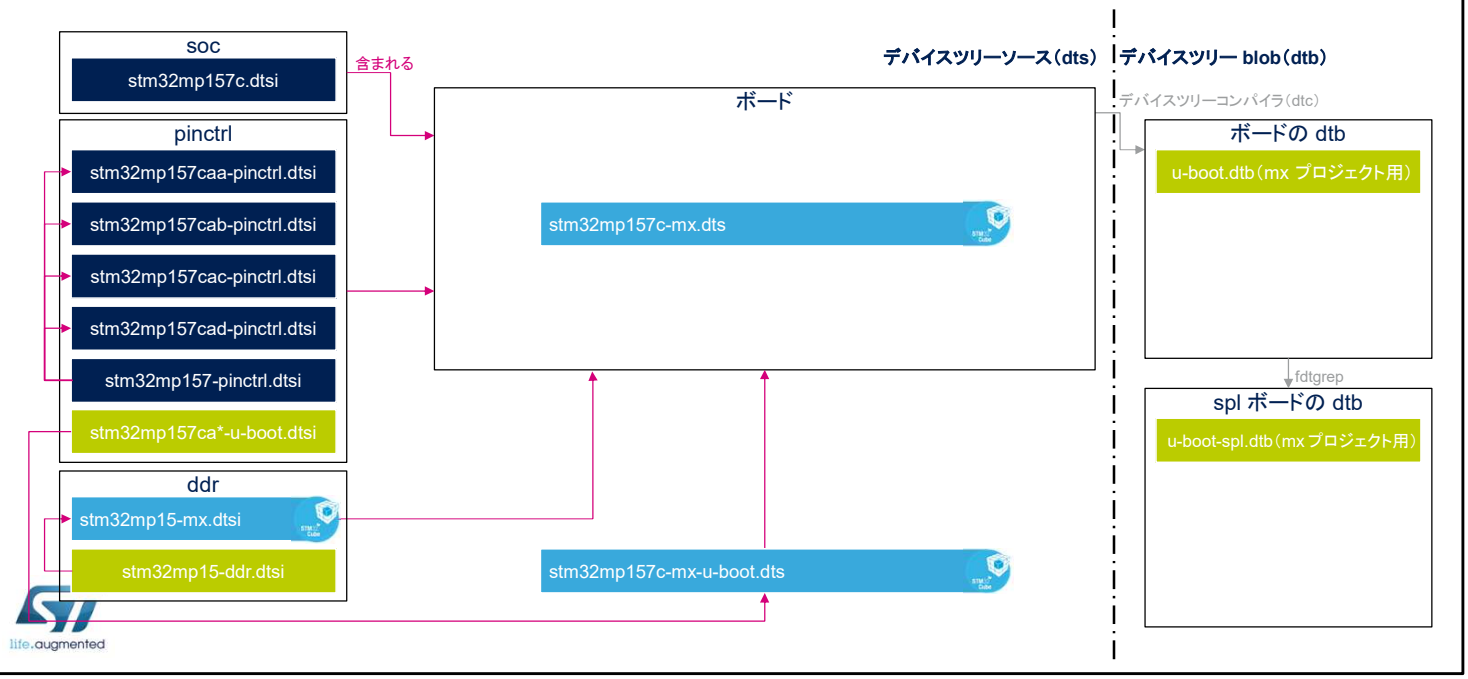
Linux



前のスライドで説明したように、STM32CubeMX ツールを使用して STM32MP1 デバイスツリーファイルを生成できます。Linux デバイスツリーファイルの生成から始めて、詳しく見ていきましょう。
STM32CubeMX は Linux 用の「ボード」ファイルのみを生成することに注意してください。これには、一方の「soc」ファイルと他方の選択されたパッケージに対応する「pinctrl」ファイルが含まれます。

STM32CubeMX による生成

U-Boot : Linux ファイルのコピーをオーバロード

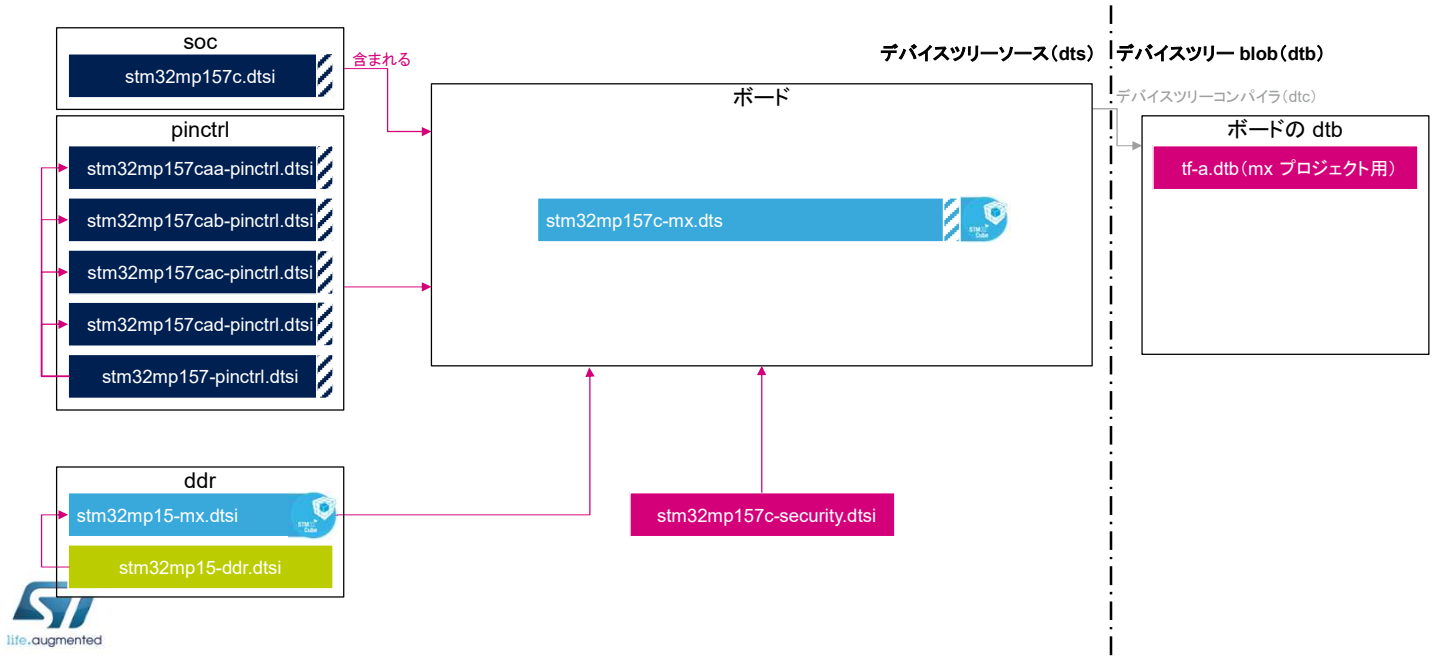


STM32CubeMX では、U-Boot の Linux dts ファイルをコピーし、次の 2 つの新しいファイルによってこれを完成させます。「ddr」構成用の 1 つのファイルと、必要な場合はいつでも「u-boot,dm-spl」および「u-boot,dm-pre-reloc」プロパティを主に使用して構成される U-Boot アドオン用のもう 1 つのファイルです。

このアプローチは、デバイスツリーファイルのアップロードバージョンに使用された開発プロセスに非常に近いものです。

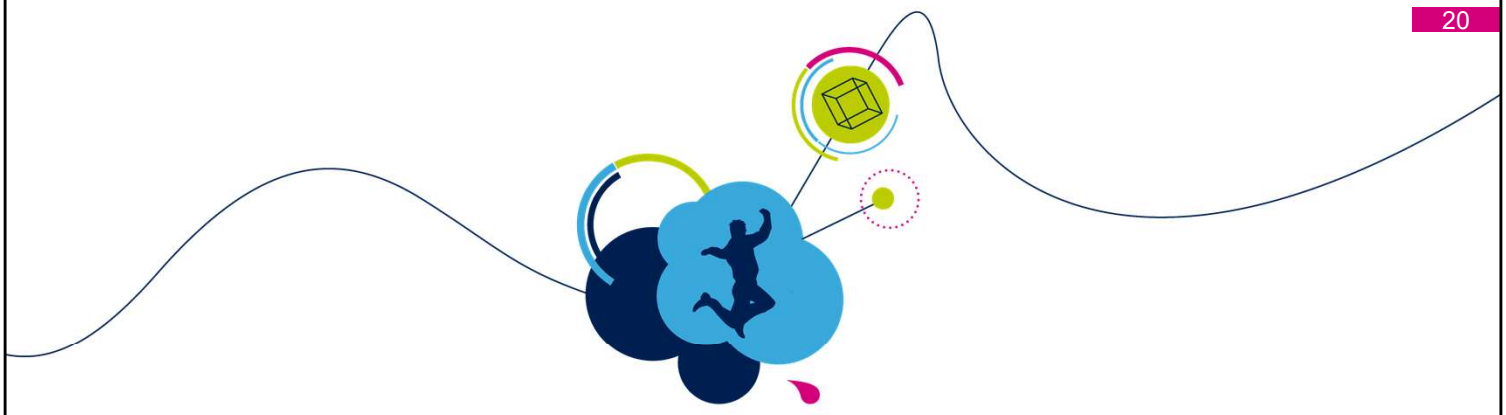
STM32CubeMX による生成

TF-A：Linux ファイルのサブセットをオーバーロードと、U-Boot からの DDR 構成を使用



STM32CubeMX は、スペースを節約するために、Linux の「ボード」dts ファイルの軽量バージョンである TF-A 用の「ボード」dts ファイルを生成します。このファイル自体には、TF-A に付属する「soc」および「pinctrl」側のすでに軽量の dtsi ファイルバージョンが含まれています。

次に、U-Boot 用に生成された同じ「ddr」構成ファイルが TF-A 用に再利用されます。



OpenSTLinux Flash メモリマッピング

正しくブートするには、ブートピンまたは OTP ブートソースを介して適切なブートデバイスを選択する必要があります。次に、選択したブート Flash を適切にパーティション分割して、ROM コードがそこからブートできるようにする必要があります。
このセクションでは、ROM コードが、選択した Flash 内の FSBL バイナリを検索する方法を示します。また、ST マイクロエレクトロニクスの OpenSTLinux ディストリビューションに実装されている完全な Flash メモリマッピングについても説明します。

Flash パーティション（最小）

サイズ	コンポーネント	コメント
残りの領域	userfs	ユーザファイルシステムにはユーザデータと例を含まれています。
768MB	rootfs	Linux ルートファイルシステムには、すべてのユーザ空間バイナリ（実行可能ファイル、ライブラリなど）とカーネルモジュールを含まれています。
16MB	vendorfs	このパーティションは、サードパーティ所有権のバイナリを配置し、GPL v3 などのオープンソースライセンスによって汚染されないようにするために、rootfs よりも優先的に使用されます。
64MB	bootfs	ブートファイルシステムには下記のものが含まれています。 - （オプション）init ram ファイルシステム。外部 RAM にコピーして、より重い rootfs をマウントする前に Linux で使用できます。 - Linux カーネルデバイスツリー（フラット化されたイメージツリー - FIT にすることができる） - Linux カーネル U-Boot イメージ（フラット化されたイメージツリー - FIT にすることができる） - NOR 以外のすべての Flash の場合：U-Boot で表示されるブートローダのスプラッシュ画面の画像 - U-Boot 配布 config ファイル extlinux.conf（フラット化されたイメージツリー – FIT にすることができる）
2MB	ssbl	セカンドステージブートローダ（SSBL）は U-Boot であり、デバイスツリー blob（dtb）が末尾に追加されています。
256～512kB（*）	fsbl	ファーストステージブートローダーは Arm のトラステッド･ファームウェア（TF-A）または U-Boot セカンダリプログラムローダー（SPL）で、末尾にデバイスツリー blob（dtb）が追加されています。少なくとも 2 つのコピーが組み込まれています。 注：ROM コードの RAM ニーズにより、FSBL ペイロードは 247kb に制限されています。 デバイスツリー blob（dtb）が、最後に追加少なくとも 2 つのコピーが組み込まれています。 注：ROM コードの RAM ニーズにより、FSBL ペイロードは 247kb に制限されています。

life.augmented

（*）：パーティションサイズは Flash テクノロジーに依存し、NOR（256kB）／NAND（512kB）のブロック消去サイズに合わせる。

ブートチェーンが考慮する順序と一致するように、この表を下から上に読んでみましょう。

- 「fsbl」パーティションには、選択されたブートチェーンに応じて、FSBL バイナリ、つまり TF-A または U-Boot SPL が含まれます。繰り返しになりますが、このバイナリには、ブートローダが使用する dtb ファイルも含まれています。
- 「ssbl」パーティションには SSBL が含まれているため、U-Boot とその dtb ファイルが含まれています。
- 「bootfs」はブートファイルシステムであり、下記のものが含まれています。
  - ❖ U-Boot の配布構成ファイル
  - ❖ スプラッシュ画面の画像、ただし NOR Flash の場合は除きます（次のスライドを参照）
  - ❖ Linux カーネルの U-Boot イメージ
  - ❖ Linux カーネルのデバイスツリー
  - ❖ オプションで、このパーティションには、起動時に Linux カーネルによって使用される可能性がある initramfs を含めることができます。

  このトレーニングの範囲外ですが、フラット化されたイメージツリー形式によって、bootfs ではこれらのバイナリの複数のバージョンを 1 つのイメージに合併することができます。
- 「vendorfs」ファイルシステムは、「rootfs」で使用される GPL v3 などの歓迎されないライセンスによって汚染されないように、サードパーティのバイナリを格納するために使用されます。
- 「rootfs」ファイルシステムにはすべてのユーザ空間バイナリが含まれるため、主にカーネルモジュール、実行可能ファイル、およびライブラリが含まれます。これは通常最大のパーティションであり、最大 800MB の幅にすることができます。
- 「userfs」ファイルシステムには、ユーザデータと ST マイクロエレクトロニクスの例が含まれています。

Flash パーティション（オプション）

サイズ	コンポーネント	コメント
256kB（*）	logo	このパーティションには、NOR Flash でのブート時のブートローダのスプラッシュ画面の画像が含まれます（他のすべての Flash の場合、画像は bootfs パーティションに保存されます）
256～512kB（*）	teeh	OP-TEE ヘッダ
256～512kB（*）	teed	OP-TEE ページング可能コードおよびデータ
256～512kB（*）	teex	OP-TEE ページャ

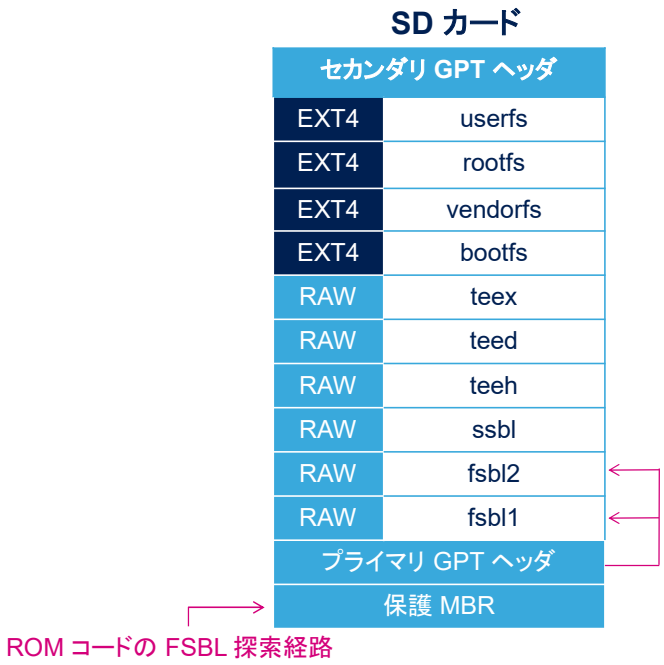
（*）：パーティションサイズは Flash テクノロジーに依存し、NOR（256kB）／NAND（512kB）のブロック消去サイズに合わせる。

life.augmented

以下のいくつかのパーティションはオプションです。

- 「teex」、「teed」、および「teeh」には、STM32MP1 プラットフォームでサポートされているセキュア OS である OP-TEE と呼ばれる オープンで移植可能な信頼できる実行環境 (Open Portable Trusted ExecutionEnvironment)が含まれています。
- 「logo」には、NOR Flash 上でのみ、ブートローダースプラッシュ画面の画像が含まれます。これは、bootfs が SD カードなどの別のデバイスに保存されており、U-Boot がスプラッシュ画面を表示しようとする時にはまだ初期化されていないからです。

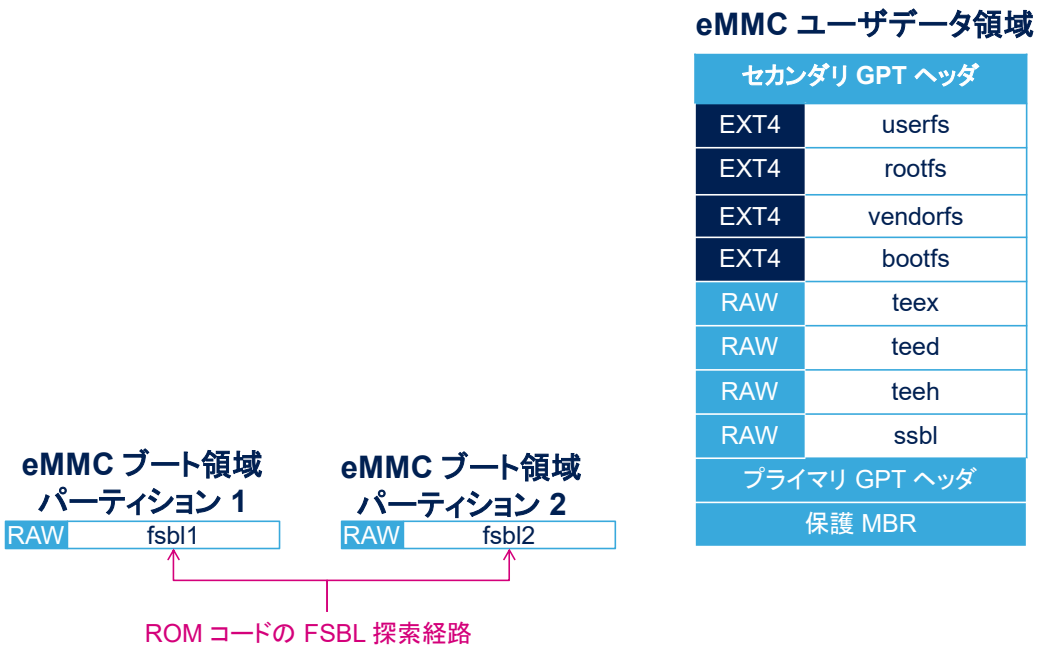
SD カードのメモリマッピング



SD カードを使用してブートしている間、ROM コードは、他のすべてのパーティションの位置を特定するために、デバイスの先頭で GUID パーティションテーブル（GPT）を探します。

- FSBL バイナリのフェイル・セーフアップデートを実行できるように、RAW パーティションで 2 つの「fsbl」コピーが使用できます。フェイル・セーフとは、「fsbl1」の更新中に電源障害が発生した場合でも、「fsbl2」パーティションに有効なコピーが存在していることを意味し、その逆も同様です。
- FSBL はファイルシステムをサポートしていないため、「ssbl」および「tee」バイナリは RAW パーティションに格納されます。
- 「bootfs」は、U-Boot がサポートするファイルシステムである EXT4 パーティションとして保存されます。
- 「rootfs」と「userfs」は、Linux で使用される EXT4 ファイルシステムです。

eMMC のメモリマッピング

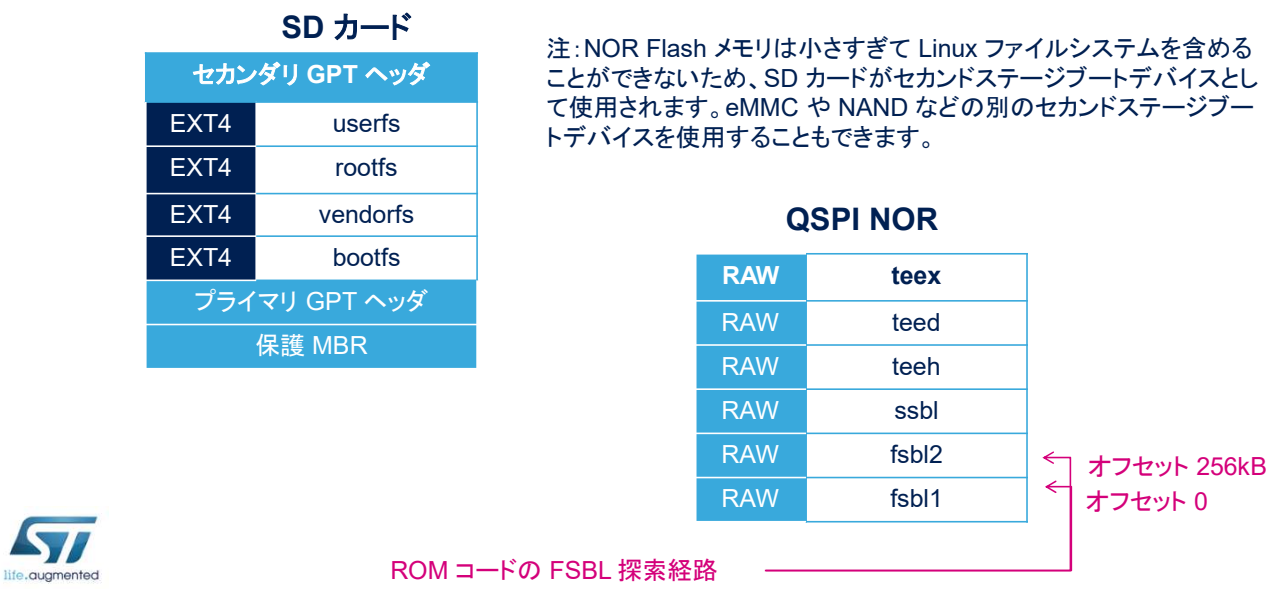


eMMC は、次のいくつかの追加の特別な物理パーティションを除いて、SD カードのように見えます。ブートコンテキストでは、「ブート領域パーティション 1」と「ブート領域パーティション 2」が重要なパーティションです。

したがって、使い方としては、1 つの「fsbl」インスタンスを「ブート領域パーティション 1」に配置し、もう 1 つのインスタンスを「ブート領域パーティション 2」に配置します。次に、ROM コードは直接、これらのブート領域パーティションの 1 つで FSBL バイナリを見つけようとします。

また、「ユーザデータ領域」は、いくつかのパーティションを見つけられるように、すべての連続したブートコンポーネントによって使用される GPT テーブルでパーティション化されます。

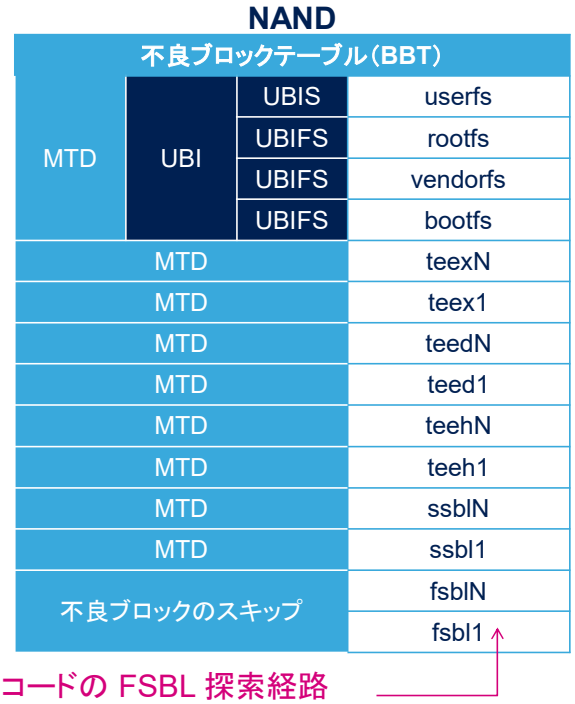
NOR メモリマッピング



コスト上の理由から、NOR Flash メモリのサイズは製品ではそれほど大きくないはずです。ほとんどの場合、小さなパーティションを配置するには十分ですが、より大きなファイルシステムを配置するには明らかに十分ではない 8MB としましょう。そのため、これらのファイルシステムを格納するために 2 番目の Flash メモリデバイスが必要です。図は、SD カードで対応するマッピングを示していますが、この 2 番目の Flash メモリは、たとえば NAND Flash の可能性があります。

NOR Flash からブートする場合、ROM コードはオフセット 0 および 256kB で「fsbl」インスタンスを探します。

NAND メモリマッピング



注:SSBL および OP-TEE パーティションは、FSBL がサポートしている場合、UBI フォーマットに移行する可能性あり。

注:[不良ブロックのスキップ] 領域では、製品の予想寿命とファームウェア更新手段に応じて、STM32CubeProgrammer の Flash レイアウトでコピー数とマージンを定義する必要あり。

NAND Flash メモリは、既存の安価な Flash テクノロジーですが、管理が最も複雑でもあります。

安いと言うことは、広く使用されることを意味しますが …

… 他方では、NAND Flash の物理的な構成と、取り扱いをより難しくしている以下の 2 つの主要な欠陥を理解することが重要です。

- 物理的な側面では、NAND Flash は「ブロック」に分割され、それ自体が「ページ」に分割されます。各ページには、データが保存される「ユーザ領域」とメタデータを保存するために使用される「スペア領域」が含まれています。
- 最初の種類の欠陥:一部のブロックが不良である可能性があります。それらのいくつかは、時間が経つにつれて製品寿命の間に、損耗のために不良になりますが、それらのいくつかは、生産時点からすでに不良になっています。工場での不良ブロックを識別するために、スペア領域で特別なタグが使用されています。
- 2 番目の種類の欠点:電子的なリークや読み取り障害などの物理現象により、一部のビットがページ内で反転することがあります。したがって、この問題を克服するために、エラー検出および訂正メカニズムが必要であり、エラー訂正コードをスペア領域に格納しています。

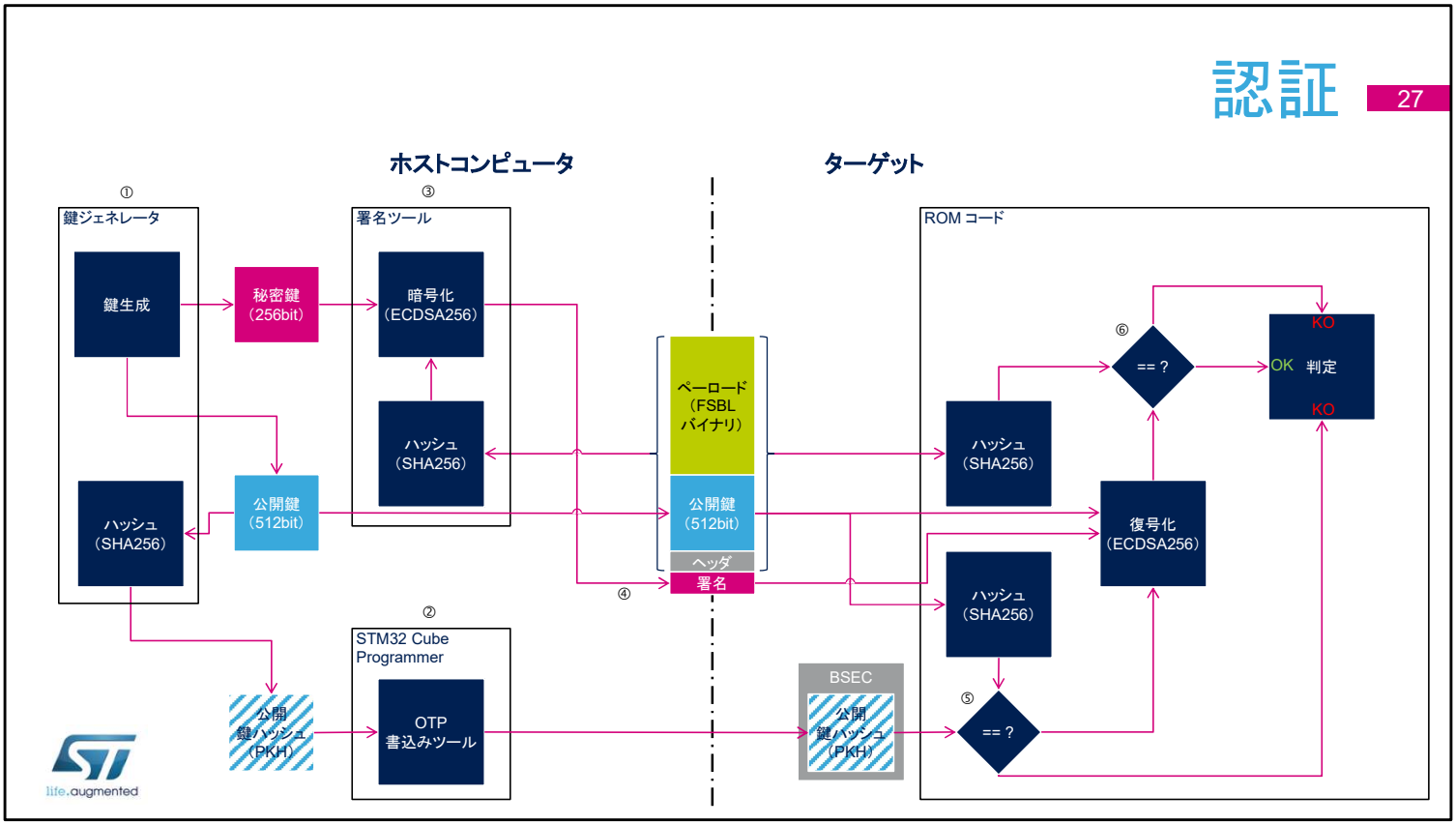
それでは、NAND Flash を管理するソフトウェアについて考えてみましょう。

- ソフトウェアは、ページ内のエラーを検出して訂正できなければなりません。これは必須です。
- 不良ブロックの交換使用を実施できないソフトウェアは、「不良ブロックをスキップする」方法を使用します。これは、不良ブロックが検出されたときに次のブロックにジャンプすることです。
- 不良ブロック管理の第 1 レベルが MTD、すなわちメモリテクノロジーデバイスに備わっています。これによって MTD ボリューム内の不良ブロックの管理が可能になります。しかし、MTD では損耗の問題を軽減することはできません。これが、MTD パーティションに含まれるバイナリには複数のコピーが必要である理由です。
- NAND Flash の究極の管理レベルは「ソートされていないブロックイメージ」(UBI)によって提供されます。UBI では、論理ブロックを介して完全なボリュームを上位層に公開します。UBI が有効な物理ブロックへの変換を管理し、ウェアレベリングの処理を行うため、UBI ボリュームおよび内部で定義されている UBI ファイルシステム内でデータを複製する必要はありません。

さてこれで、上の図を以下のように理解することができます。

- ROM コードは「不良ブロックのスキップ」戦略を使用して、NAND Flash のオフセット 0 から始まる有効な fsbl インスタンスを探します。
- ssbl および tee パーティションは、個別の MTD ボリューム内に複数のコピーとして格納されます。
- bootfs、rootfs、および userfs は、共通の MTD ボリューム内の UBIFS パーティションとして定義されます。

認証



このスライドは、STM32 MPU で使用されるイメージの署名と認証のプロセスの概要です。
前提条件として、正しい理解のために知っておくべき最小限の略語を以下に示します。

- ペイロードは、暗号化操作が実行されるバイナリファイルです。
- ECDSA256 は、256bit の鍵ペアを持つ楕円曲線デジタル署名アルゴリズムの略です。これは、256bit の秘密鍵によってペイロードを暗号化し、対応する 512bit の公開鍵によってそれを復号化できる非対称暗号アルゴリズムです。
- SHA256 は、入力ペイロードから 256bit のハッシュを生成するセキュアハッシュアルゴリズムを意味します。
- 後で公開鍵の整合性をチェックするために公開鍵ハッシュを保存することは、一般的な方法です。
- 署名は、認証する初期ペイロードと公開鍵を含むペイロードの暗号化ハッシュで構成されています。署名はしばしば HMAC（Hashed Message Authentication Code: ハッシュ化されたメッセージ認証コードの略）と呼ばれます。

詳細については、暗号化の関連文書とトレーニングを参照してください。

この図では、ROM コードによって FSBL バイナリの認証を行うために導入されている以下の完全な流れを説明しています。

1) まず、ホスト側で「鍵ジェネレータ」ツールを使用して ECDSA 鍵のペア、すなわち「秘密鍵」と対応する「公開鍵」を生成します。また、このツールによって、SHA256 操作を介して「公開鍵ハッシュ」も生成します。
2) 「STM32CubeProgrammer」には、「公開鍵ハッシュ」を STM32MP1 BSEC 不揮発性メモリに書き込むために使用できる「OTP 書込みツール」が含まれています。
3) 「署名ツール」は、ホスト側で FSBL ペイロードと「公開鍵」およびファイル「ヘッダ」の SHA256 ハッシュを計算するために使用されます。
4) このハッシュは、ペイロードの「署名」を得るために「秘密鍵」を使用して ECDSA256 で暗号化されます。「署名ツール」は、「ペイロード」、「公開鍵」、「ヘッダ」、「署名」を含む署名済みファイルを最終的に生成します。この署名済みファイルは、「STM32CubeProgrammer」が内蔵フラッシュに実装するために使用するファイルです。
5) STM32MP1 ターゲットがリセットされると、ROM コードは、署名されたファイル内にある「公開鍵」の SHA256 ハッシュを計算することを始めます。次に、このハッシュを STM32MP1 BSEC 不揮発性メモリに保存されているハッシュと比較します。異なる場合、認証プロセスは失敗します。そうでない場合、次に進みます …
6) ROM コードは、FSBL ペイロードと「公開鍵」およびファイル「ヘッダ」の SHA256 を計算します。このハッシュを、認証されたばかりの「公開鍵」を使用して復号化された「署名」から得られた値と比較します。比較が失敗した場合、認証プロセスは失敗します。そうでない場合、認証は成功し、ROM コードはブートプロセスを続行します。