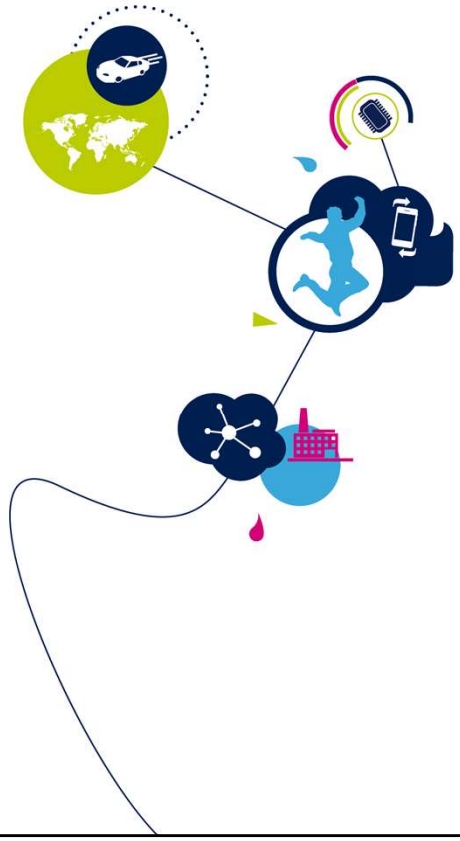
STM32MP1 - COPROC

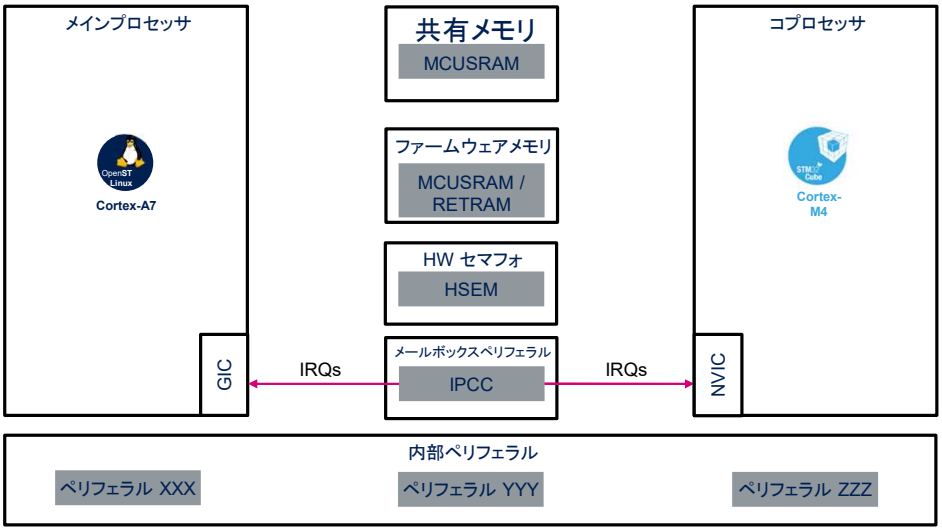
コプロセッサ管理

1.0 版



こんにちは、STM32MP1 コプロセッサ管理のプレゼンテーションへようこそ。ここでは、プロセッサ同士がコマンドとイベントを介して相互作用し、データを交換し、リソースを共有する方法について説明します。

システムの概要



STM32MP1 マルチプロセッサシステムでは、独立したファームウェアを 2 つの CPU コアで実行できます。

- マスタの Arm Cortex-A7 プロセッサは Linux ベースの OS を実行するために最適化されています。
- スレーブ（またはコプロセッサ）の Arm Cortex-M4 プロセッサは、マイクロコントローラまたはベアメタルアプリケーション向けに最適化された RTOS を実行できます。

さらに、いくつかの内蔵された STM32 ペリフェラルをこれらのプロセッサのいずれかに割り当てることができます。

次に、2 つの内部メモリ領域がマスタプロセッサとスレーブプロセッサ間で共有されます。これらのメモリは、コプロセッサのファームウェアをロードして実行するのに使用されますが、プロセッサ間通信用の共有バッファなどの共通的構造を定義するのにも使用されます。
さらに、プロセッサ間通信の場合、プロセッサ間通信コントローラ（IPCC）という名前の特定のペリフェラルにより、専用メールボックスによる信号の授受が可能になります。
最後に、ハードウェアセマフォ（HSEM）を使用して、共有リソースを同時アクセスから保護できます。

主な概念

- コプロセッシング管理は、異種の非対称アーキテクチャでマルチコアを取り扱うために導入されたメカニズムです。
- コプロセッサを管理するために、以下のサービスのリストが提案されています。
  - Cortex®-A7（Linux または Uboot）による Cortex-M4 ファームウェアのロードと制御
  - RPMsg メッセージとメールボックスに基づくプロセッサ間通信
  - リソース管理：
    - コアへのペリフェラルの割当て（アクセス権要求による）
    - システムリソース管理（両コアに共通のリソース）

life.augmented

詳細に入る前に、マルチプロセッサシステムでのコプロセッサの管理に関連する概念を理解することが重要です。
1 つ目は、Cortex-M4 コアのロードと制御です。Cortex-A7 コアは、Cortex-M4 ファームウェアのロードと Cortex-M4 コアのリセットの制御を担当します。
その後、共有メモリと IPCC ペリフェラルに依存した RPMsg プロトコルによってプロセッサ間通信が確保されます。
最後に、次の機能を提供するリソース管理サービスが提案されます。

- ペリフェラルをコアに割り当て、このコアのみによるこのペリフェラルへの排他的アクセスを制御します。
- たとえばクロックや GPIO など、2 つのコア間で共有される共通（またはシステム）リソースへのアクセスを管理します。

関係するフレームワーク

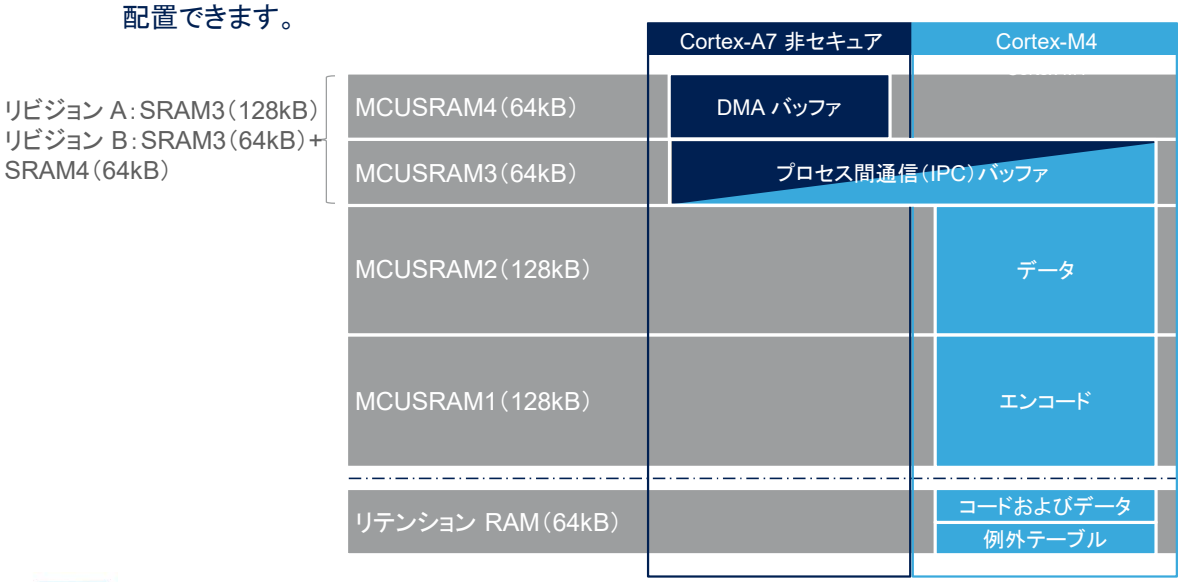
- Cortex-A7 では、Linux OS により、リモートプロセッサの制御と、それとの通信を可能にする下記フレームワークが統合されています。
  - **RemoteProc** - 汎用のリモートプロセッサフレームワークにより、リモートプロセッサの制御（電源オン、ファームウェアのロード、電源オフ）が可能になります。
  - **RPMsg** - リモートプロセッサメッセージングは VirtIO ベースのメッセージングバスであり、これによって、カーネルドライバがシステムで利用可能なリモートプロセッサと通信できるようになります。
  - **VirtIO** - VirtIO フレームワークは仮想化をサポートします。これにより共有リングバッファ（Vring）に基づく効率的なトランスポート層が提供されます。
- Cortex-M4 では、**OpenAMP** は、ベアメタルおよびリアルタイム OS（freeRTOS、Zephyr など）用の RemoteProc および RPMSg フレームワークを統合しているライブラリです。

life.augmented

ソフトウェアに関しては、コプロセッサの管理は 3 つの主要なフレームワークを通して行われます。
リモートプロセッサすなわち RemoteProc フレームワークは、Cortex-M4 コプロセッサが適切に動作するために必要なファームウェアとすべてのリソースをロードする役割を果たします。
RPMsg という名前のリモートプロセッサメッセージングフレームワークは、プロセッサ間通信に使用され、VIRTual Input Output すなわち Virtio [Virt] [I] [O] フレームワークに依存しています。
この VIRTual Input Output フレームワークは、Vring [V] [ring] という名前の共有リングバッファプールを管理するメカニズムを提供します。
Cortex-A7 では、これらのフレームワークは Linux カーネルディストリビューションでネイティブです。RemoteProc フレームワークは U-Boot にも組み込まれており、Linux ファームウェアの前に Cortex-M4 ファームウェアをプリロードできます。
Cortex-M4 では、ベアメタルおよびリアルタイム OS 用の OpenAMP [Open] [AMP] ライブラリを組み込むことによって、フレームワークを利用できます。

共有 RAM メモリのマッピング

- Cortex-M4 ファームウェアは RETRAM のアドレス 0 から開始する必要がありますが、部分的に MCUSRAM に配置できます。



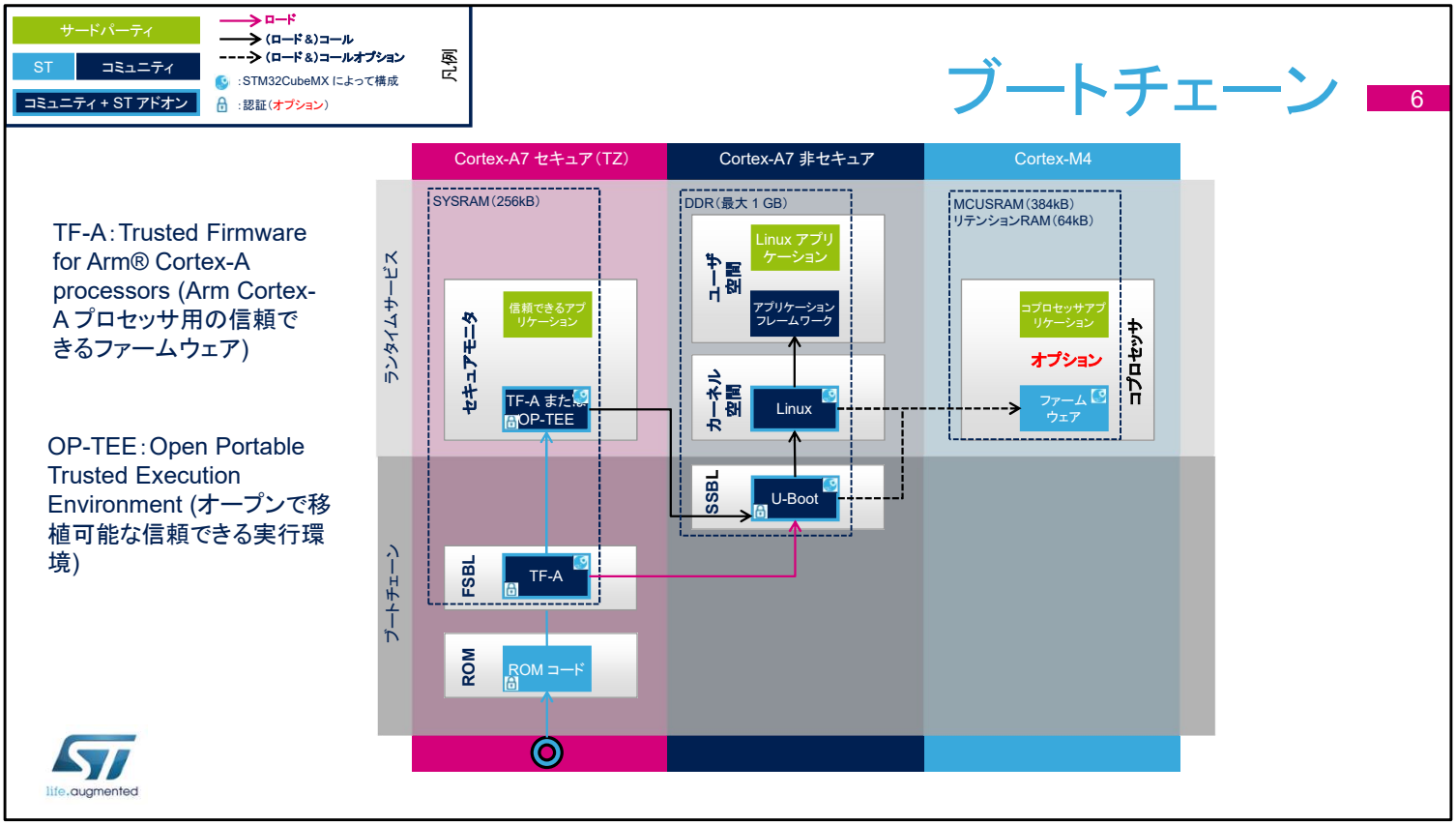
- もちろん、お客様は、製品のニーズに合わせてこのマッピング（領域の場所とサイズ）を調整できます。

このスライドは、STM32MP1 デバイスで利用可能ないくつかの RAM バンクと、STM32 MPU の組込みソフトウェアディストリビューションに適用される一般的なマッピングを示しています。

- RETRAM は、Cortex-M4 がベクタテーブルとコードおよびデータを配置するために使用します。
- MCU SRAM1 および SRAM2 を使用して、Cortex-M4 ファームウェアの残りのコードとデータを配置できます。
- MCU SRAM3 は通常、プロセス間通信バッファ（詳細は後述）のマッピングに使用されます。
- MCU SRAM4 は、DMA1 または DMA2 インスタンスを使用するときに広バンド幅が必要な場合に、Cortex-A7 用の DMA バッファを配置するために予約できます。

このマッピングを領域の境界に合わせことは必須ではありませんが、Cortex-M4 メモリのハードウェア分離がバンク単位の粒度でサポートされているため、これは大きな関心事になる可能性があります。

ブートチェーン



信頼できるブートチェーンは、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。
Cortex-M4 ファームウェアは、Linux OS またはセカンダリブートローダ(U-Boot など)によってロードし、開始できます。
«通常»モードでは、ファームウェアはファイルシステムに保存されており、RemoteProc ファイルシステムインタフェースを介して Linux ユーザランドによってロードされます。
«早期ブート»モードでは、ファームウェアは BootFS パーティションにインストールされ、Linux ファームウェアが起動する前に U-Boot によってロードされます。

ファームウェアの形式

7

- ファームウェアは ELF 形式ファイルに保存されています。
- ファームウェアバイナリには、オプションのリソーステーブルが、両方の Cortex コアから認知される特定の「.resource_table」セクションに定義されています。このテーブルは、以下の共通リソースを定義するために使用されます。
  - デバッグのためにコプロセッサのトレースにアクセスするためのユーザ sysfs インタフェースを提供します。
  - メッセージングサービスをサポートするために RPMsg および VirtIO フレームワークをロードします。

```
/* Resource table with trace and Vring for IPC */
struct remote_resource_table __resource __attribute__((used)) rproc_resource = { ¥

                .version = 1, ¥
                .num = 2, ¥
                .reserved = {0, 0}, ¥
                .offset = { ¥
                                offsetof(struct remote_resource_table,
                                        rpmsg_vdev),
                                offsetof(struct remote_resource_table,
                                        cm_trace), ¥
                }, ¥
                /* Virtio device entry for IPC */
                .rpmsg_vdev= {
                                RSC_VDEV, VIRTIO_ID_RPMSG_, 0,
                                RPMSG_IPU_C0_FEATURES, 0, 0, 0,
                                NUM_VRINGS, {0, 0},
                },
                /* Vring rsc entry - part of vdev rsc entry */
                .rpmsg_vring0 = {VRING_TX, VRING_ALIGN, VRING_SIZE, 1, 0},
                .rpmsg_vring1 = {VRING_RX, VRING_ALIGN, VRING_SIZE, 2, 0},

                /* trace buffer declaration accessible from Cortex-A7*/
                .cm_trace = {
                                RSC_TRACE,
                                (uint32_t)system_log_buf,
                                sizeof(system_log_buf), 0, "cm4_log",
                },
};
```

life.augmented

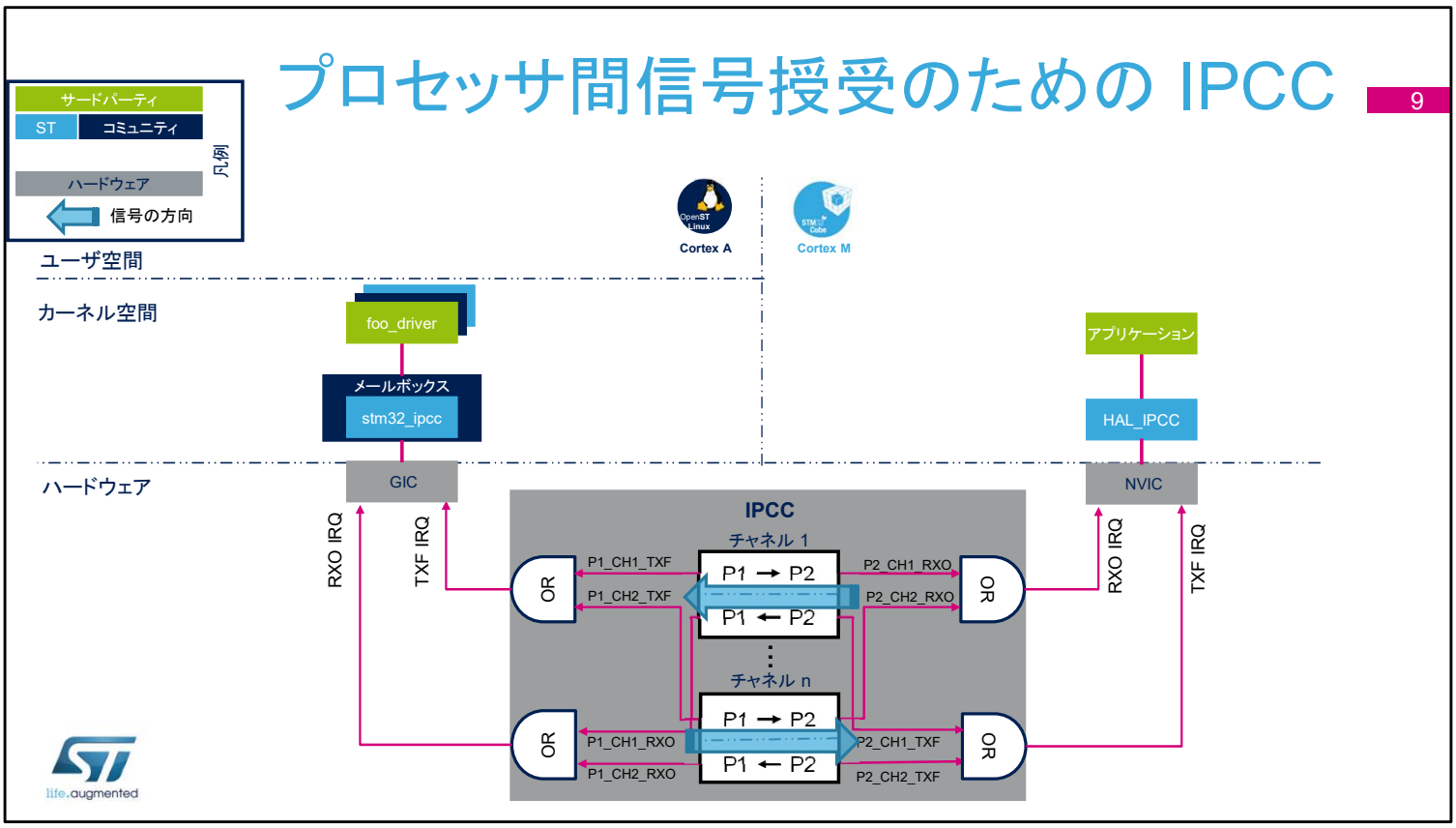
Cortex-M4 ファームウェアは ELF 形式で保存されており、U-Boot または Linux から RemoteProc フレームワークによってロードされます。
コードおよびデータセクションに加えて、コプロセッサ管理に関連する機能をサポートするために特定のセクションを定義することができます。このセクションは、ファームウェアのロードフェーズ中に RemoteProc Linux フレームワークによって解析されるリソーステーブル構造で構成されています。
このテーブルで宣言されている最初の機能は、Cortex-M4 ログをリングバッファに出力する可能性があるリモートプロセッサトレースバッファです。このバッファは Cortex-A7 側で監視できます。リングバッファのアドレスとサイズはリソーステーブルで宣言されます。
2 つ目は、プロセッサ間通信プロトコルです。RPMSG プロトコルを有効にするには、Virtio デバイスおよび関連する virtio リングディスクリプタをテーブルで宣言する必要があります。Cortex-A7 RemoteProc フレームワークは、マスタとして機能し、関連する RPMsg バッファを MCUSRAM に割り当て、結果としてこのテーブルを完成させます。

プロセッサ間通信

- プロセッサ間通信とは、Cortex-A コアと Cortex-M コア間の通信を可能にするために準備された下記メッセージングサービスを指します。
  - メッセージサービスは、共有メモリを介して RPMsg および Virtio フレームワークによって管理されます。
  - ドアベル／メールボックスサービスは、IPCC 内部ペリフェラルによって提供されます。

Cortex-M4 プロセッサと Cortex-A7 プロセッサ間のプロセッサ間通信は、RPMSg メッセージングサービスに依存しています。RPMsg および Virtio フレームワークは、通信に関連するバッファの管理を担当します。
次に、ドアベルまたはメールボックスメカニズムを使用して、新しいメッセージが利用可能であることをプロセッサに通知します。この信号は、プロセッサ間通信コントローラ（IPCC）によって生成されます。

プロセッサ間信号授受のための IPCC



STM32MP1 に統合された IPCC ペリフェラルは、Cortex-A7 と Cortex-M4 間の通信用に 6 つの双方向チャネルを提供します。
原理は、次のとおりです。

- コア、たとえば Cortex-A7 は、他のコア、ここでは Cortex-M4 に Rx 占有（RXO）割込みを生成するチャネルフラグを設定します。
- Cortex-M4 はフラグをクリアしてチャネルを空けます。これによって Cortex-A7 に TX フリー（TXF）割込みが生成されます。

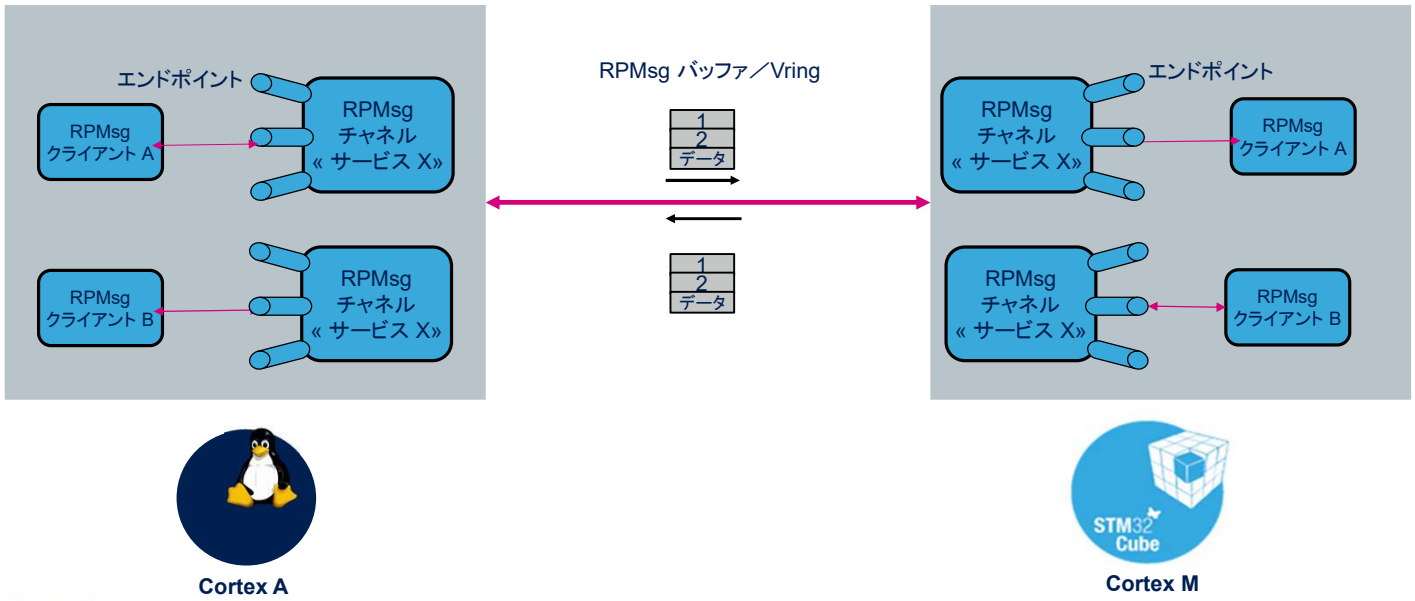
IPCC ペリフェラルはバッファを管理せず、ドアベルのようにしか機能しないことを理解することが重要です。

RPMsg メッセージングは、2 つの双方向チャネルを使用します。チャネルは、RPMSg バッファが使用可能であることを通知するために一方向で使用され、RPMSg バッファが解放されたことを通知するために他の方向で使用されます。STM32 MPU 組込みソフトウェアディストリビューションでは、IPCC チャネル 1 は Cortex-M4 から Cortex-A7 コアへのメッセージに使用され、IPCC チャネル 2 は Cortex-A7 から Cortex-M4 コアへのメッセージに使用されます。

Cortex-A7 では、IPCC はメールボックスソフトウェアフレームワークによって制御されます。

Cortex-M4 では、IPCC は HAL_IPCC ソフトウェアドライバによって制御されます。

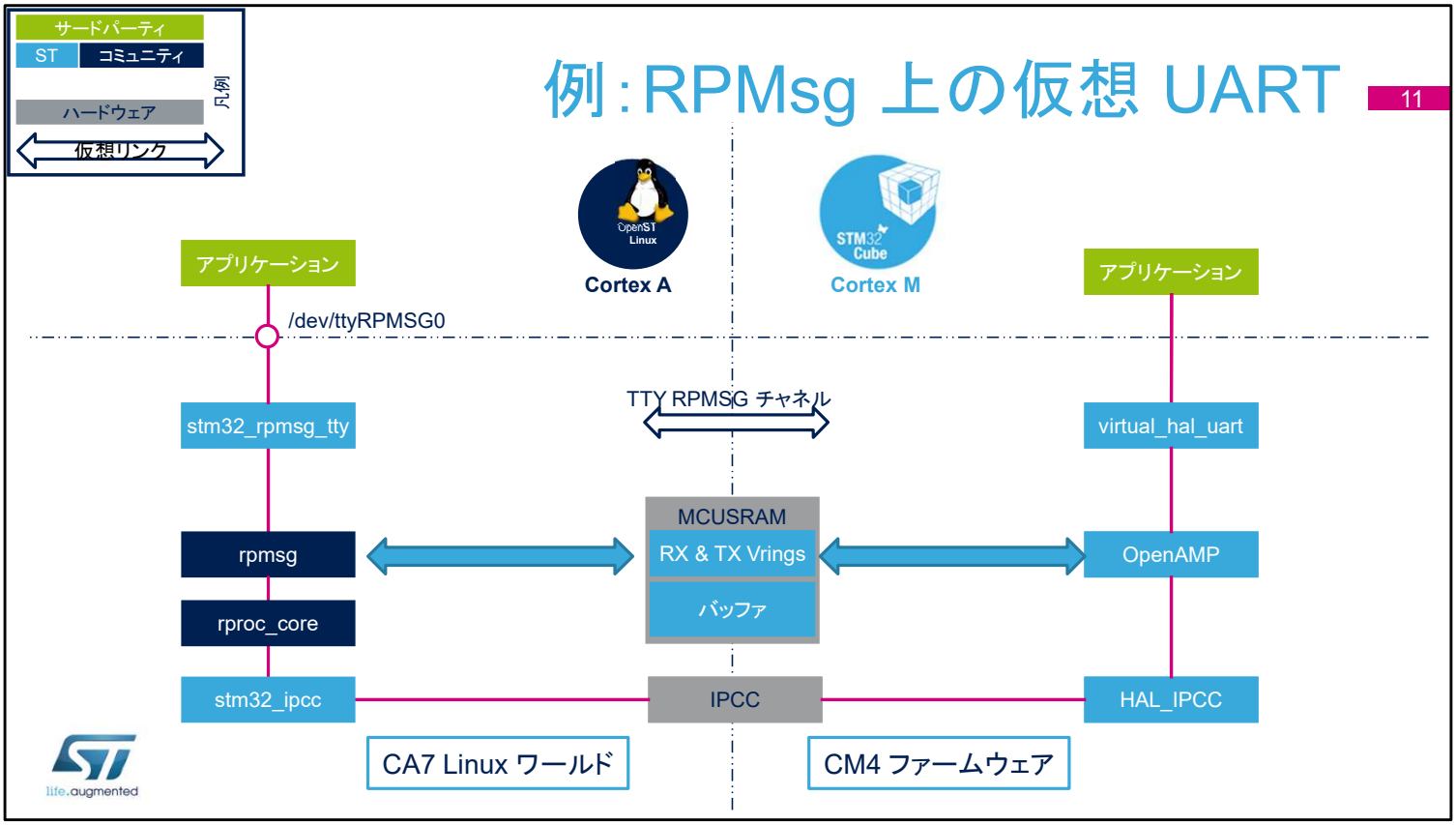
RPMsg プロトコル



STM32 MPU 組込みソフトウェアディストリビューションでは、RPMsg プロトコルがプロセッサ間通信に使用されます。
このスライドは、RPMSg プロトコルとそれに関連する概念のプレゼンテーションです。
RPMSG プロトコルは、ローカルアドレスからリモートアドレスへのメッセージの送信で構成されます。メッセージは共有メモリのバッファに格納されます。Virtio 層は、Vrings という名前の、関連ディスクリプタを持つリングバッファを使用して、バッファライフサイクルの管理を担当します。
それぞれの Cortex で、RPMsg クライアントがサービスを提供します。このクライアントは、下記によって識別されます。

- サービス名称によって定義されるサービス
- アドレス識別子と操作コールバックによって定義されるエンドポイント

RPMsg フレームワークは、サービスの登録時に、«ネームサービスのお知らせ» メッセージをリモートプロセッサに送信します。このメッセージでクライアントエンドポイントのアドレスが提供されます。
リモートプロセッサ側では、RPMSG フレームワークは、ローカルクライアントが同じサービスを登録しているかどうかを確認します。この場合、両方の RPMsg クライアント間にチャネルが作成され、ローカル RPMsg クライアントにチャネルが結合されていることが通知されます。



この仮想 UART は、RPMsg プロトコルを使用するアプリケーションを示すために STM32MP1 デバイス用に実装した例です。ここでの目的は、RPMSg プロトコルで UART をシミュレートすることです。
Cortex-A7 コアでは、stm32_rpmsg_tty ドライバは RPMsg クライアントであり、Linux ユーザランドにシリアル TTY コンソールを公開し、UART と RPMsg プロトコル間のアダプテーション層を実現するために実装されています。
Cortex-M4 コアでは、「仮想 HAL UART」サービスが RPMsg クライアントです。アプリケーション向けに HAL UART API を提供し、UART と RPMsg プロトコル間に同じアダプテーション層も提供しています。
Cortex-M4 上の仮想 UART の初期化によって RPMSG チャネルが作成されます。これにより、「TTY RPMSG」サービスのエンドポイントが作成され、「新しいサービス」のお知らせが Cortex-A7 に送信されます。

Cortex-A7 側では、メッセージが処理され、対応するサービスが stm32_rmpsg_tty ドライバに関連付けられます。Linux ドライバが探査されてエンドポイントが作成され、/dev/ttyRPMSG0 ファイルシステムインタフェースが作成されます。これ以降、Cortex-A7 コアは仮想 UART チャネルを介してメッセージを送信できます。

仮想 UART の複数のインスタンスを作成できることに注意してください。RPMSG プロトコルで言えば、このアクションはインスタンスごとに新しいエンドポイントを作成することになります。その結果、/dev/ttyRPMSG<X> インタフェースが Cortex-A7 側に作成されます。

用語

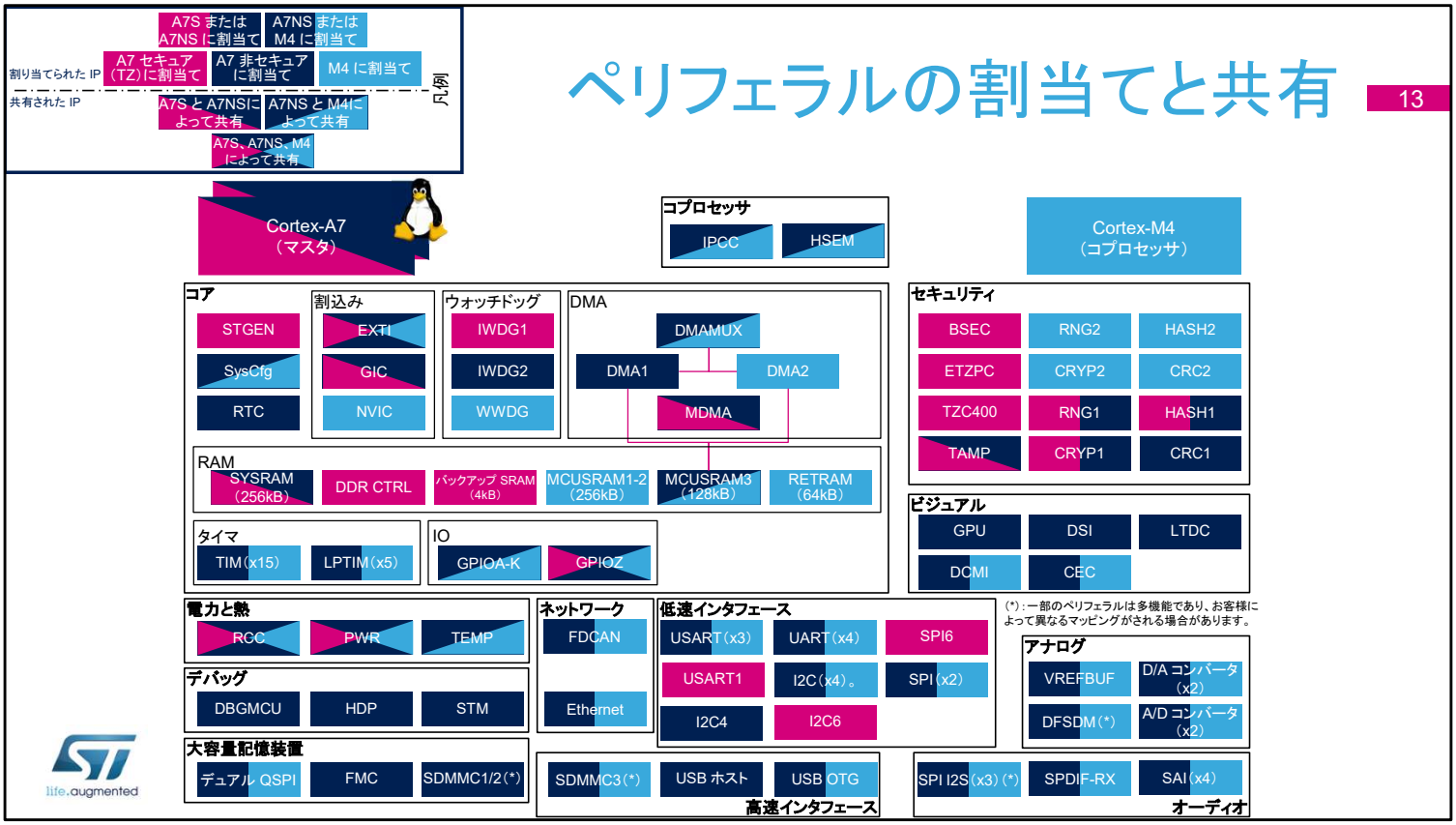
12

- STM32 のリソース管理メカニズムを理解するために、知っておくべきいくつかの以下の用語があります。
  - **割当て**:コアへのペリフェラルの割当ては、このペリフェラル(またはハードウェアブロック)がそのプロセッサによって使用されることを意味します。
  - **共有リソース**:ペリフェラルの動作に必要な共有リソースまたはシステムリソースであり、ペリフェラルが割り当てられているマスタコアによって制御されます(クロック、レギュレータ、GPIO などのリソース)。

life.augmented

コプロセッサのリソース管理は、マルチコアシステムでのペリフェラルの管理メカニズムの実装に対応しています。
まず、リソース管理を理解するために、いくつかの概念を定義する必要があります。

ペリフェラル割当てという用語は、1 セットのペリフェラルを Cortex のコンテキストに割り当てるアクションを意味します。たとえば、I2C ペリフェラルは Cortex-A7 コアまたは Cortex-M4 コアのどちらかに割り当てることができます。
他方で、いくつかのペリフェラルまたはリソースは、いくつかのコンテキストで共有する必要があります。共有リソースは通常、リセットおよびクロック制御用の RCC のようなシステムリソースです。



このスライドでは、STM32MP1 プラットフォームで可能なペリフェラル割当てを示しています。この概念に沿って、この図の凡例は、各ペリフェラルの以下の 2 つの可能な状態を区別して示しています。

- 「割り当てられた」とは、1 つのハードウェア実行コンテキストのみが実行時にペリフェラルを使用していることを意味します。単一色のボックスは割当てが静的であることを意味しますが、垂直セパレータ付きの 2 色または 3 色のカラーボックスは、アプリケーションのニーズに応じて、特定のコンテキストにペリフェラルを割り当てるためにユーザの選択が必要であることを意味します。この割当ては、STM32CubeMX ツールで、または手動で行うことができ、Cortex-A7 セキュアおよび Cortex-M4 コアのハードウェア分離によって補強されています。
- 「共有された」とは、ペリフェラルが同時に 2 つまたは 3 つでさえも異なる実行コンテキストによって使用できることを意味します。このモードは、共通リソースにアクセスするときに、指定されたコンテキスト間で競合が発生しないことを保証するレジスタバンキングや他のメカニズムを使うことを意味します。

例:

- 左側にある TIM インスタンスは 1 つのランタイムコンテキストに割り当てることができ、このコンテキストのみによって使用されます。
- すぐ下の RCC ブロックは、3 つのランタイムコンテキストから同時にアクセスできるシステムペリフェラルです。

以下のスライドの目的は、「割り当てられた」リソースと「共有された」リソースを管理するために STM32MP1 プラットフォームに実装された概念を説明することです。

割当てに関連するすべてのメカニズムは、ST Wiki ページでさらに説明されています。
この図は、STM32 MPU 組込みソフトウェアディストリビューションにおける割当ての ST マイクロエレクトロニクスの推奨または選択を示していることに注意してください。その他の可能性については、STM32MP1 のリファレンスマニュアルで説明されている可能性があり、このディストリビューションで後で検討される場合があります。

ペリフェラル管理の課題

14

- ペリフェラルを Cortex-M4 コアまたは Cortex-A7 コアに割り当てる:
  - ペリフェラル(またはハードウェアブロック)がそのプロセッサによってのみ使用されることを意味します。
  - 開発者は、グローバルシステムでの割当ての一貫性に責任があります。
  - STM32CubeMX ツールでは、TF-A、U-Boot、Linux、および HAL_init のデバイスツリーに Cortex-A7 または Cortex-M4 が使用することを事前入力することで、開発者のためにこの一貫性を保証することができます。
- 分離:
  - ペリフェラルは、セキュアな部分または Cortex-M4 コア用に分離できます。
  - TF-A セキュアファームウェアは、ETZPC レジスタに基づくペリフェラルの分離をデバイスツリーに従って管理します。
  - STM32CubeMX ツールでは、Uboot/TF-A のデバイスツリーに事前入力することで、開発者のためにこの分離を実現できます。
- クロックや GPIO などの共有リソースを設定する
  - リソースは、ハードウェアセマフォによって保護されるか、デバイスツリーに従って Linux によって設定されます。
  - STM32CubeMX ツールでは、Linux のデバイスツリーおよび HAL_init 関数に事前入力することで、開発者のためにこのリソース設定を行います。

life.augmented

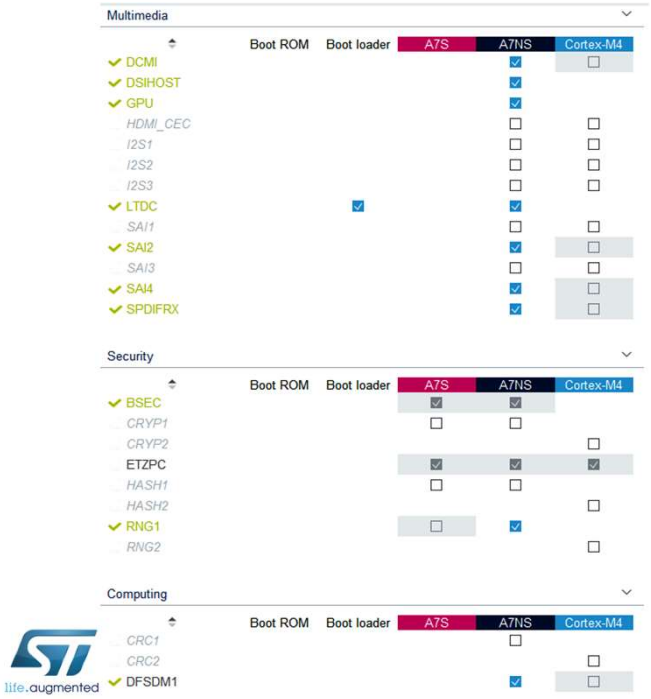
リソース管理の実装を理解するには、マルチプロセッサ環境で発生する関連課題を理解することが重要です。

STM32MP1 マイクロプロセッサでは 2 つのコンテキストが並行して動作しています。ペリフェラルが 1 つのコンテキストに割り当てられている場合、このペリフェラルはこのコンテキストのみが管理しなければなりません。したがって、グローバルシステムの一貫性を確保する必要があります。これを実現するために、STM32CubeMX ツールは、ペリフェラルを割り当てるためのインタフェースを提供しています。

STM32MP1 は、セキュアな部分または Cortex-M4 にペリフェラルを分離する可能性も提供しています。分離は、拡張 TrustZone 保護コントローラ(ETZPC)テーブルの設定を介して、セキュアファームウェア TF-A によって管理されます。STM32CubeMX ツールでは、分離に関係する TF-A デバイスツリーを設定することでこれを管理することもできます。

最後に、一部のリソースは共有リソースです。これらのリソースは、一般にペリフェラルを操作するために必要になります。そのタイプに応じて、システムリソースは、システムの一貫性を確保するために、ハードウェアセマフォで保護するか、Linux コンテキストで排他的に管理する必要があります(たとえば、クロックツリー管理)。同様に、STM32CubeMX ツールは、共有リソースを正しく設定するのに役立ちます。

STM32CubeMX によるペリフェラルの割当て



- チェックボックスにチェックを入れると:
  - ペリフェラルを次のコンテキストの 1 つに分離するために TF-A デバイスツリーにコードが生成されます。
    - Cortex-A7 セキュア
    - Cortex-M4
    - Cortex-A7 非セキュア
  - Cortex-A7 またはCortex-M4 でドライバを有効にするためにコードが生成されます(デバイスツリー)。
  - Linux デバイスツリーにコードが生成され、ペリフェラルの設定に応じてペリフェラルの動作に必要なすべての共有リソース(クロック、レギュレータ、GPIO、EXTI)が宣言されます。
  - Cube ファームウェアコードが、Cortex-M4 コアに関連するペリフェラルを初期化するために生成されます。

このスクリーンショットは、STM32CubeMX ツールで実行できる割当ての例を示しています。

- USART4 は、Linux 用に Cortex-A7 非セキュアコンテキストに割り当てられています。
- USART5 は、STM32Cube 用に Cortex-A4 コンテキストに割り当てられています。

選択した設定に応じて、STM32CubeMX ツールは、次の方法でペリフェラルを Cortex コンテキストに割り当てます。

- TF-A のデバイスツリーを設定することでペリフェラルを分離する。
- 割当てに応じてノードを有効または無効にする Linux デバイスツリーを設定する。
- Cortex-M4 コアに割り当てられたペリフェラルを操作するために必要なシステムリソースを宣言する。
- Cortex-M4 コアに割り当てられたペリフェラル用の Cube ファームウェア初期化コードを生成します。

リソース管理サービス

- 2 つの実行コンテキストの共存を保証するために、リソース管理によって以下のいくつかのサービスが提供されます。
  - ペリフェラル割当てリクエスト：Cortex-M4 で使用されるメカニズムで、ペリフェラルがプロセッサ用に予約されていることを確認します。原則は、使用を開始する前にペリフェラルの可用性を確認することです。
  - コプロセッサ共有リソース構成の設定：コプロセッサでペリフェラルを操作するために必要な共有リソースを構成するために、メインプロセッサ（Linux を実行する Cortex-A7）で利用可能なサービス。
  - ハードウェアセマフォによる共有ペリフェラルの保護：両方のプロセッサで利用できるこのサービスにより、GPIO と EXTI の共有リソースを同時アクセスから保護できます。
  - 共有リソースの動的再構成：コプロセッサがメインプロセッサに共有リソース構成の更新を要求できるようにするリモートメッセージングに基づくサービス。

リソース割当ての制御とチェックを支援するために、いくつかのサービスが STM32 MPU 組込みソフトウェアディストリビューションに実装されています。

- 最初のサービスは、STM32Cube ファームウェアに実装されたペリフェラル割当てチェックです。このユーティリティは ETZPC テーブルに依存しており、初期化前にペリフェラルの分離を確認できます。
- 2 つ目のサービスは、共有構成の設定です。Cortex-A7 コアに実装され、ペリフェラルに関連付けられたクロックとレギュレータの設定を可能にします。これらの共有リソースは、電源管理の最適化のために Linux OS によって管理される必要があります。
- 次に、各 Cortex ファームウェアが GPIO および EXTI リソースの設定を担当します。これらの共有リソースの保護は、HSEM ペリフェラルに依存するハードウェアセマフォの使用によって保証されます。
- 最後に、リソースの動的再構成サービスが導入され、コプロセッサが動作中にクロックとレギュレータの設定を変更できるようになりました。

ペリフェラル割当てリクエスト

- ETZPC テーブルに基づく設定:

  ブート段階で、信頼できるファームウェア TF-A はペリフェラル割当てを次のように設定します。

  - TF-A はブートローダのデバイスツリーから設定を読み取ります。
  - TF-A はそれを ETZPC レジスタへ書き込みます。

設定	A7 Linux アクセス	M4 ファームウェア アクセス
セキュア	不可	不可
非セキュア、M4 分離	不可	可
非セキュア、A7 または M4	可	可
非セキュア、A7 分離 *	可	不可

* ソフトウェア分離のみ、ハードウェア分離なし(ハードウェアアクセスは可能)

- チェック:

  ペリフェラル割当てリクエスト機能により、ドライバはペリフェラルのアクセス可能性を確認できます。ドライバを開始する前に、ペリフェラルへのアクセスが以下のように許可される必要があります。

  - Cortex-A7 では:U-Boot が、ETZPC テーブルに依存する Linux デバイスツリーのペリフェラルノードを更新することにより、アクセスを許可します。
  - Cortex-M4 では:ドライバを開始する前に、ペリフェラルへのアクセスはリソースマネージャユーティリティによって許可される必要があります。

life.augmented

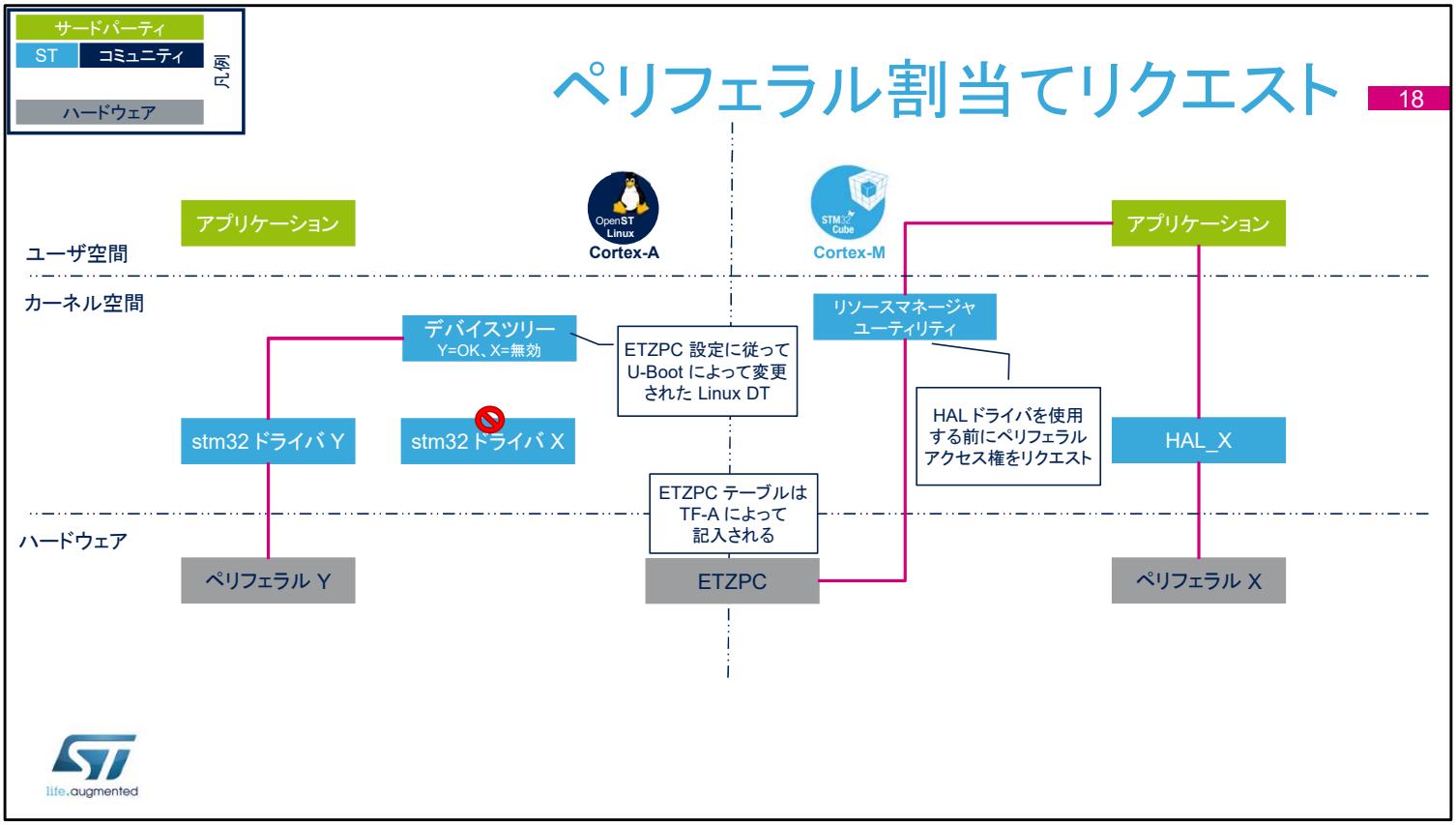
ペリフェラル割当てリクエストサービスは、拡張 TrustZone 保護コントローラ(ETZPC)テーブルによって制御されるバスファイアウォールに依存しています。
セキュア/非セキュアコンテキストと Cortex-A7 または Cortex-M4 コンテキストへの割当てに応じて、4 つのモードが利用可能です。

Cortex-M4 コアからのアクセスは引き続き可能であるため、他のモードとは逆の非セキュア Cortex-A7 の分離は、実際のハードウェア分離ではなくソフトウェア保護であることに注意してください。

ETZPC テーブルは、TF-A セキュリティファームウェアによってそのデバイスツリーに基づいて入力されます。

その後、初期化段階で、次のようにアクセスが許可されます。

- Cortex-A7 コアでは、U-Boot は ETZPC テーブルに従って Linux のデバイスツリーを更新します。割当てに応じて、Linux での使用に宣言されたペリフェラルノードを有効または無効にします。
- Cortex-M4 コアでは、アプリケーションはペリフェラルを初期化する前にペリフェラルの割当てを確認する必要があります。リソースマネージャユーティリティは、このアクセスを許可するサービスを提供します。



この図は、ペリフェラル割当てのチェックに関連するソフトウェアアーキテクチャの概要を示しています。
この例では、TF-A ファームウェアがペリフェラル X を分離して Cortex-M4 コアに割り当て、ペリフェラル Y をCortex-A7 コアに割り当てています。

- Cortex-A7 コアでは、ブート段階で U-Boot がペリフェラル Y ノードを有効にし、ペリフェラル X ノードを無効にしています。その結果、ペリフェラル Y の stm32 ドライバのみが有効になります。
- Cortex-M4 では、アプリケーションはリソースマネージャユーティリティによってペリフェラル X へのアクセス権を付与し、HAL API を呼び出して初期化します。

共有リソース構成の設定

- 共有リソースは、両方のコアがペリフェラルを操作するために必要な場合に、同時アクセスから保護する必要があります。2 つの戦略:
  - ハードウェアセマフォを使った保護
  - 1 コアだけによる共有リソースの管理
- **ハードウェアセマフォによる保護**: 共有リソースを保護してリソースレジスタへの同時アクセスを回避するデフォルトの方法
  - **ピン**:ピンの構成は Linux の pinctrl フレームワークによって管理されます。GPIO モードでは、コプロセッサーは出力値の変更、および入力値の読出しができます。
  - **外部割込み(EXTI)**:GPIO と、関連する EXTI 割込みの間の結び付けは、Linux レギュレータフレームワークによって管理されます。コプロセッサは、NVIC 割込みマスクの有効化／無効化を担当します。
- **Cortex-A7 Linux フレームワークによる管理**:電源管理戦略に関係する特定のシステムリソースに対して:
  - **クロックツリー**:クロック設定は Linux クロックフレームワークによって管理され、コプロセッサはペリフェラルのクロックゲーティングを担当します。
  - **レギュレータ**:電源は Linux のレギュレータフレームワークによって管理されます。

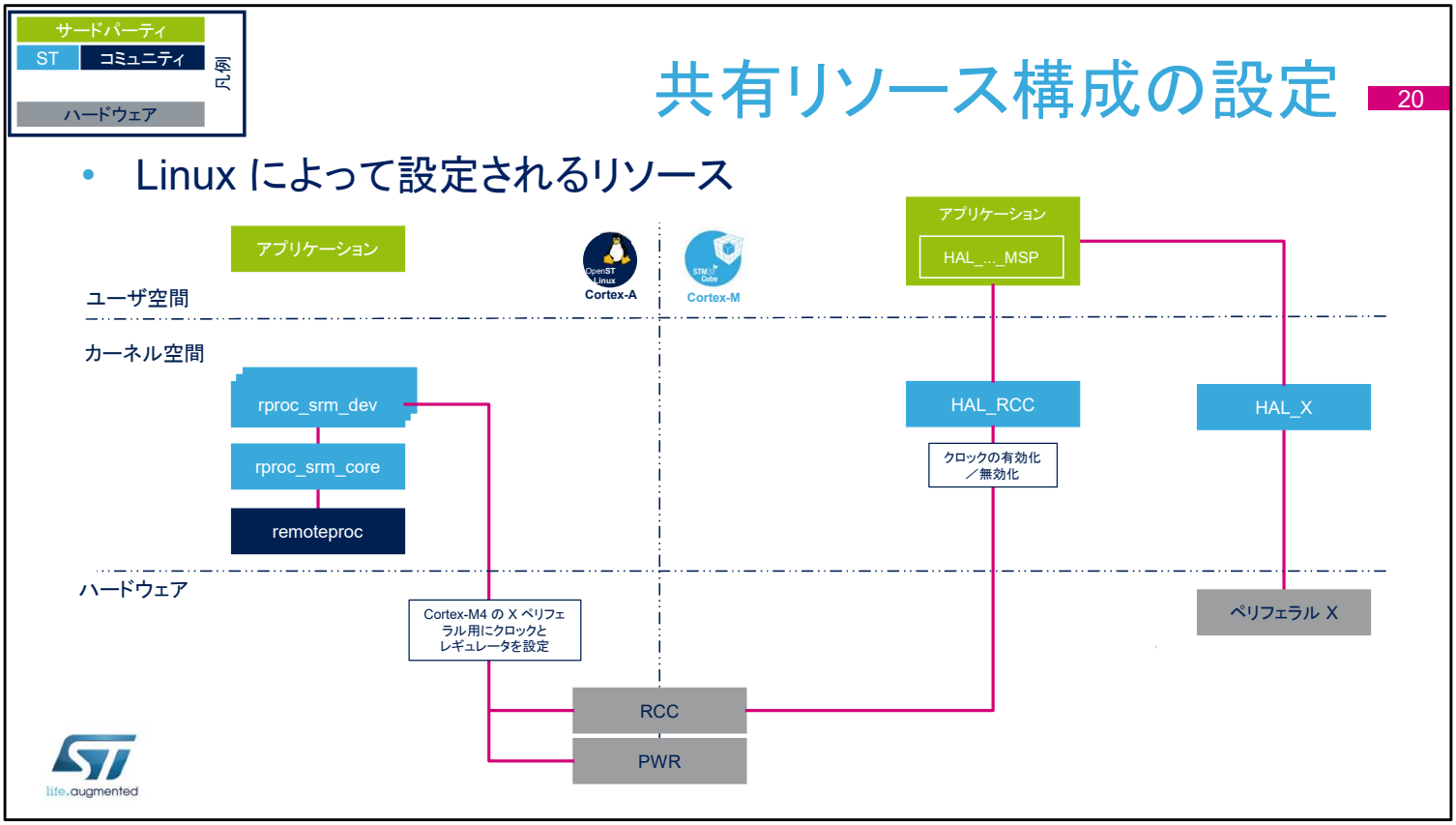
life.augmented

このサービスの主な目的は、マルチコアシステムの共有リソースの構成を正しく管理することです。
これらのリソースを保護するために、以下の 2 つの戦略が実装されています。

- ハードウェアセマフォによる共有リソースの保護
- 1 つの Cortex だけによる共有リソースの管理

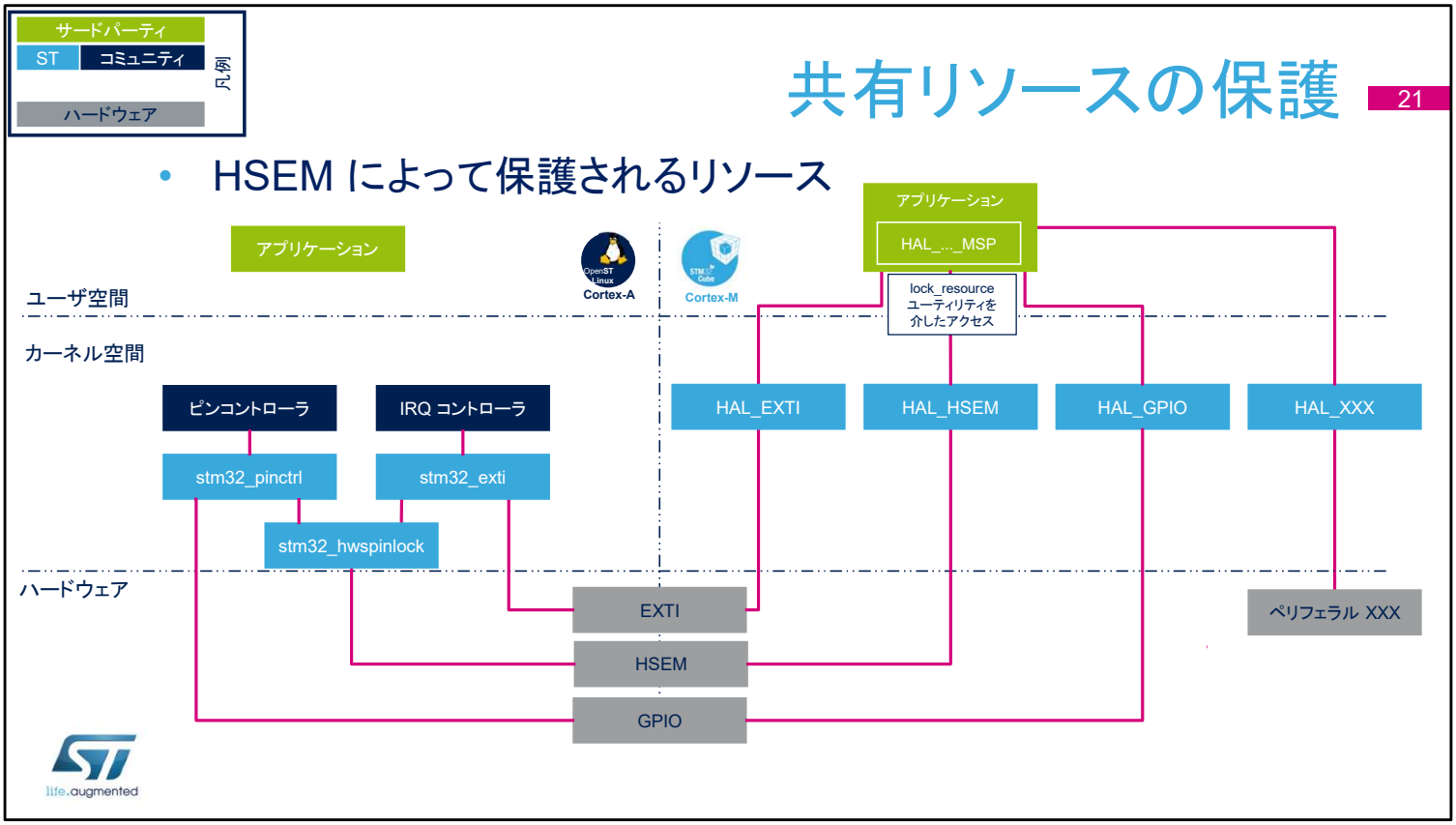
HSEM ペリフェラルに依存するハードウェアセマフォによる保護がデフォルトで使用されます。これは、ハードウェアセマフォに共有リソースの重要なレジスタへの排他的アクセスを許可させることから成ります。これは、たとえば GPIO および EXTI 構成の場合です。

クロックやレギュレータなどの他の共有リソースは、Linux OS で管理する必要があります。主な理由は、Linux OS にネイティブに実装されている統合電源戦略です。実際、クロックとレギュレータの連鎖は、親ノードと子ノードを持つツリーで表すことができます。このツリーは Linux OS によって監視され、子の状態に応じて親を無効または有効にします。たとえば、Linux は Cortex-M4 コアで使用されるクロックについて通知を受ける必要があります。この情報がない場合、Linux はクロックが使用されていないと見なし、親ノードとして PLL クロックを停止する可能性があります。



このスライドは、Cortex-A7 コア上の Linux OS による共有リソースの構成に関与するソフトウェアフレームワークを示しています。
リモートプロセッサシステムリソースマネージャ（RPROC SRM）フレームワークは、割り当てられたペリフェラルを操作するために Cortex-M4 コアが使用する共有リソースを宣言および構成するために ST によって開発されました。このフレームワークは、Cortex-M4 ペリフェラルに関連付けられているノードのデバイスツリーを解析し、それに応じて共有リソースを構成します。SRM デバイスドライバインスタンスはサブノードごとに有効にされます。

Cortex-M4 側では、クロックおよびレギュレータリソースは初期化されません。クロックゲーティングレジスタは Cortex に依存するため、ペリフェラルクロックのみが停止されます。



このスライドは、ハードウェアセマフォによる共有リソースの保護に関与するソフトウェアフレームワークを示しています。
Cortex-A7 コアでは、GPIO および EXTI の設定レジスタへのアクセスは、HSEM セマフォがフリーならば取得し、そうでなければ解放されるまで待機するハードウェアスピンロックによって保護されます。
U-Boot ブートローダもブート段階で同じハードウェアセマフォを使用することに注意してください。

Cortex-M4 コアでは、アクセス保護はアプリケーションによって管理され、ロックリソースユーティリティ API によって許可される必要があります。
このユーティリティは、保護するリソースに応じてセマフォをロック／ロック解除するシンプルなインタフェースを提供しています。

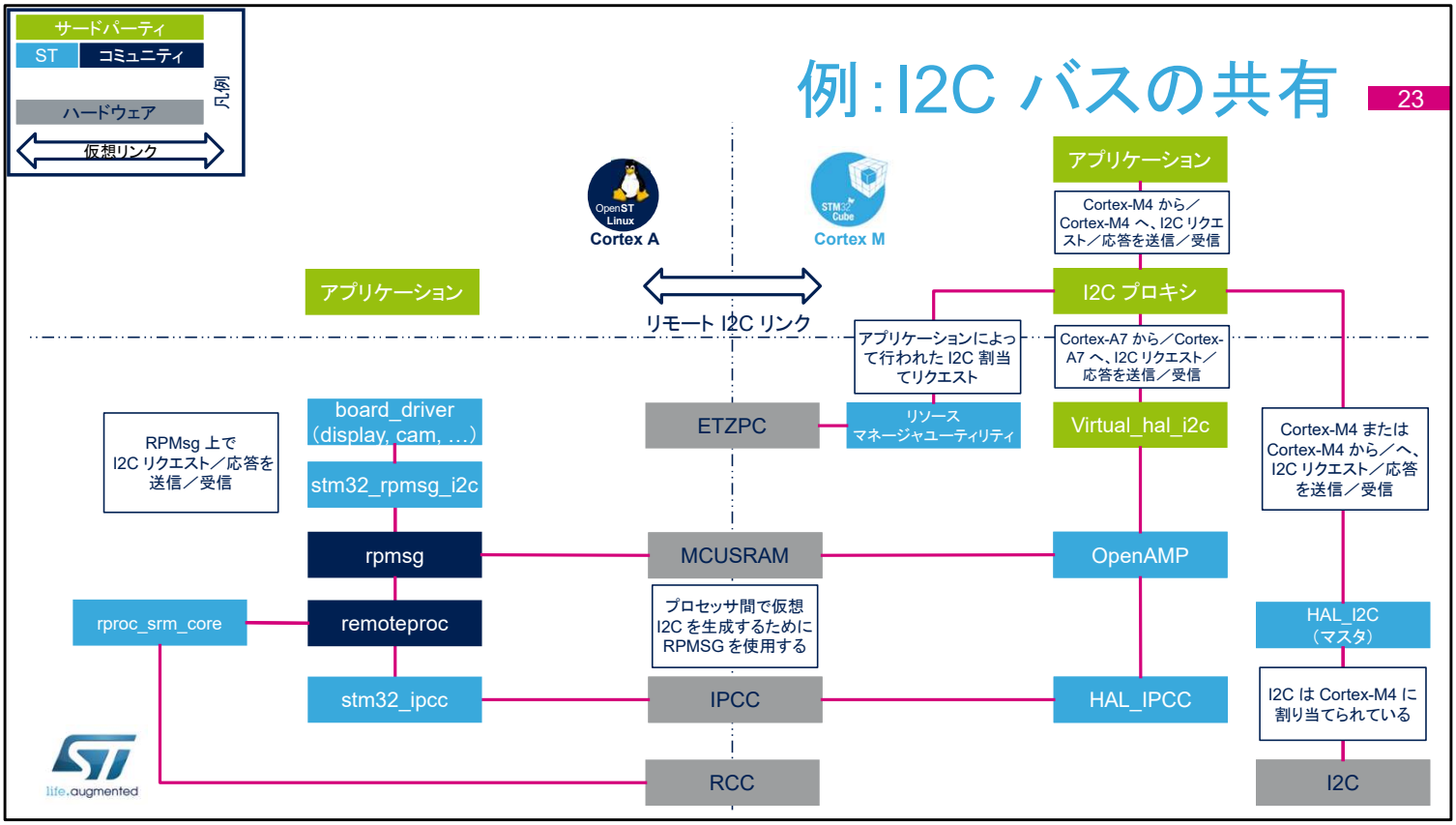
例：Cortex-M4 に割り当てられた I2C

コードの設定に関して、STM32CubeMX ツールが共有リソースを管理するためのコードを生成します。
このスライドでは、Cortex-M4 コアに割り当てられた I2C ポートの共有リソースの生成について説明します。

Cortex-A7 デバイスツリーにおいて、リソースマネージャノードが、関連するクロックと GPIO の宣言とともに定義された I2C1 サブノードで宣言されています。ペリフェラルを Cortex-A7 コアに割り当てるために使用された I2C1 ノードは以前に無効になっていることに注意してください。
I2C ポートに使用されるピンもデバイスツリーで宣言されていることに注意してください。GPIO はハードウェアセマフォによって保護されているため、この部分は任意です。ただし、この宣言はデバッグに使用することができ、ピンが別のペリフェラル用に予約されていないことを Linux OS がクロスチェックできるようになります。

Cortex-M4 側では：

- クロックの初期化は、「SystemResourceConfigAllowed」サービスによって保護されています。実際、特定のケースでは、Cortex-M4 コアがクロックツリーを初期化する必要がある場合があります。これは、Cortex-M4 が U-Boot のステージでロードされ、Linux ファームウェアの前に開始された場合に当てはまります。このサービスを使用すると、ブートモードに関係なく、同じファームウェアを使用できます。
- この例では、GPIO の設定は、ハードウェアセマフォを取得するロックサービスによって保護されています。



これは、以前に提示した概念のほとんどを使用したより複雑な例です。
STM32MP1 プラットフォーム上の Cortex-A7 コアと Cortex-M4 コアの間で I2C ポートを共有する方法について説明します。
この例では、I2C ペリフェラルは Cortex-M4 コアによって管理されています。これは、ETZPC テーブルによって I2C が Cortex-M4 用に分離され、Linux のファームウェア SRM フレームワークが I2C バスの動作に必要なクロックを設定したことを意味します。

Cortex-A7 コアと Cortex-M4 コア間で I2C フローを転送するために、STM32 MPU 組込みソフトウェアディストリビューションで提案されている仮想 UART と同じ原理で、RPMSg 上の仮想 I2C リンクを実装することができます。

Cortex-M4 コアでは、Cortex-M4 アプリケーションまたは仮想 I2C HAL から I2C リソースを共有できるようにするために、I2C プロキシが必要です。
コプロセッサ管理または ST32MP1 製品自体の詳細については、st.com で入手可能な STM32MP1 ユーザーガイドを参照してください。