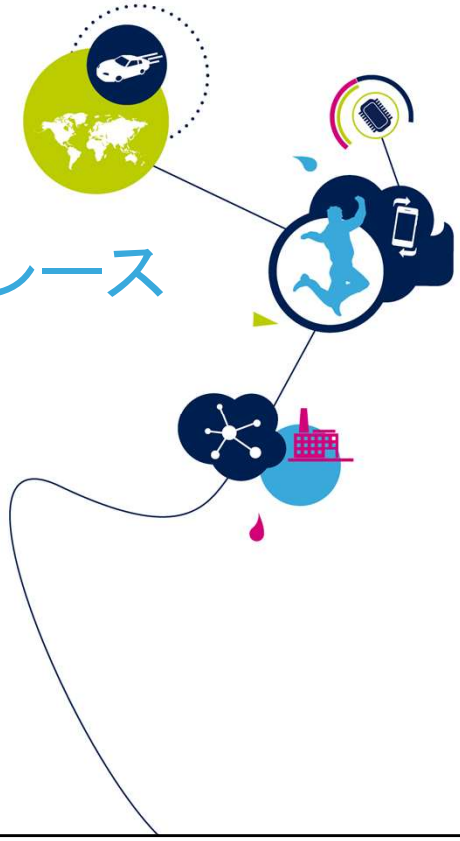
STM32MP1 プラットフォームトレースおよびデバッグのソリューション



Linux カーネルのトレース、モニタリング、デバッグ入門

1.0 版

life.augmented

STM32MP1 プラットフォームでのトレースおよびデバッグのソリューションに関するトレーニングにようこそ。
このスライドセットでは、Linux カーネルを実行する Cortex-A7 パートのみについて説明します。
Cube ソフトウェアを実行する Cortex-M4 パートについては、別のモジュール（STM32MP1-Software-Coprocessor マネージメント）で説明します。

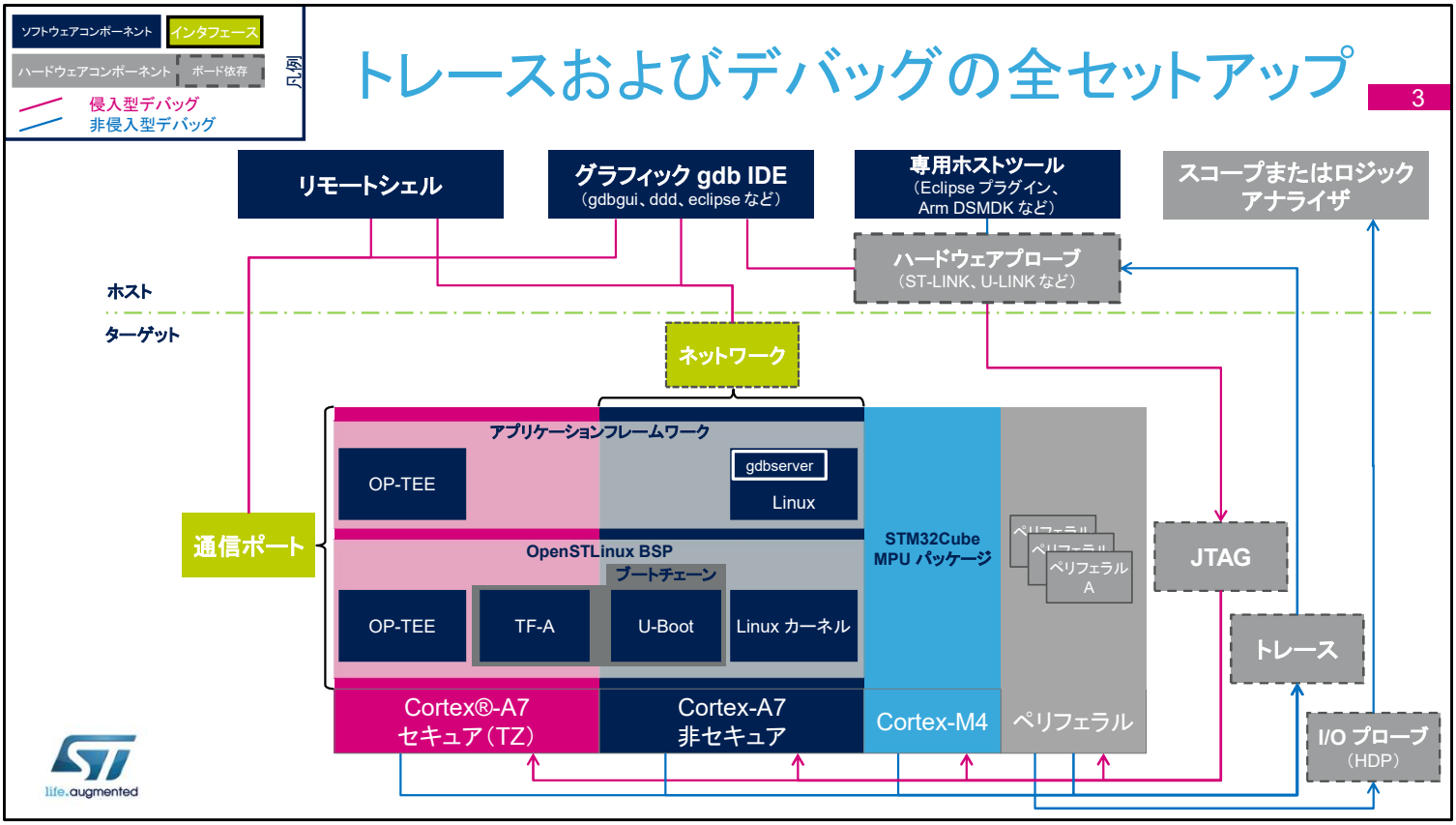
概要

- STM32MP1 プラットフォームには、さまざまなトレースおよびデバッグのパスやソリューションが準備
- STM32MP1 プラットフォームでのトレースおよびデバッグは次のいずれか
  - ハードウェア指向：JTAG ポート、ETM、STM、ITM の使用、HDP での内部信号の監視による Arm® コアのデバッグ…（**ディスカバリボードでは JTAG のみで、すべてが使用できるわけではありません。**）
    - ハードウェアプローブ（ST-LINK、U-LINK など）または専用ホストツールが必要
  - ソフトウェア指向：標準の Linux ツール（strace、perf、ftrace、kmemleak など）を使用した OS のトレースおよびモニタリング
- トレースおよびデバッグセッションは次のいずれか
  - 非侵入型デバッグ：
    - ログやトレースの読出しや収集によって、すべてのシステム動作で変更が生じることはない
  - 侵入型デバッグ：
    - ログやトレースの読出しや収集によって、システム動作に変更が生じます。
    - 使用したツールに応じて、動作の変更規模はわずかなものから無視できないものまでがある

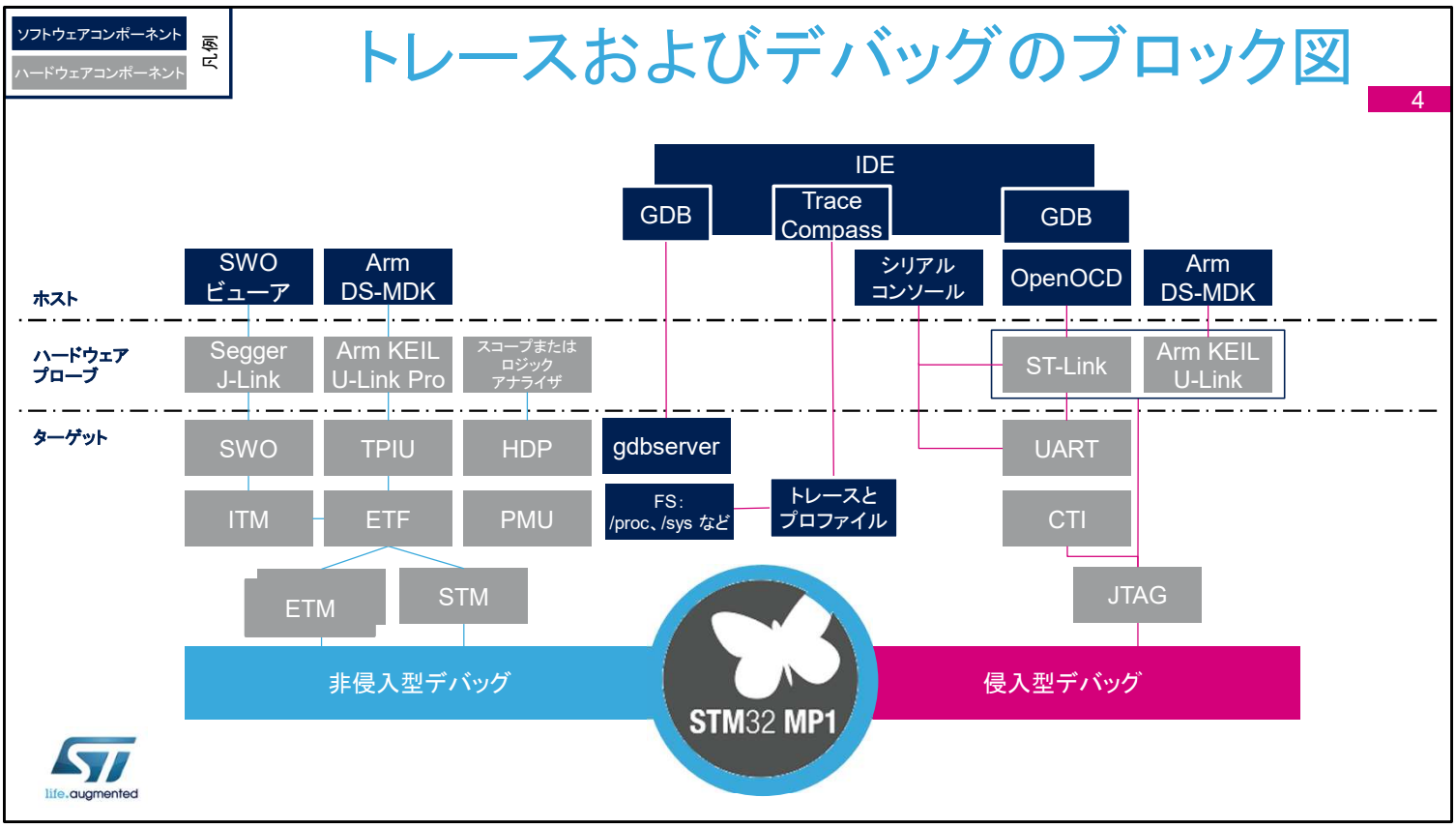
STM32MP1 プラットフォームには、対象とする部分に応じた、さまざまなトレースおよびデバッグのパスやソリューションが備わっています。
STM32MP1 プラットフォームではハードウェアとソフトウェアの両方で提示されています。

- ハードウェアパスでは、Arm コア（Cortex-A7 や Cortex-M4）のデバッグや、ハードウェアデバッグポート（HDP）を使用したペリフェラル間の内部信号の監視に、ハードウェアの信号が有効です。これらのインタフェースはすべてのボードにデフォルトであるわけではないことに注意してください（例：ディスカバリボードでは JTAG ポートのみ使用できます）。
- ソフトウェアパスでは、Linux のトレースおよびデバッグの環境とツールスイートを使用できます。一部のツール（例：strace、perf、ftrace、kmemleak など）は、Linux カーネルに組み込まれており、デフォルトで有効にする必要はありません。その他のツールは、ユーザ空間レベルでインタフェース接続すると、外部から統合できます。

デバッグパスについて、Arm は侵入型デバッグモードと非侵入型デバッグモードの概念も導入しています。
非侵入型デバッグ環境では、すべてのシステム動作に変更を加えることなく情報を読み出したり収集したりします。つまり、システムのセットアップや実行に影響を与えないということです。一方、侵入型デバッグ環境では、システム動作が影響を受けます。この影響は重要になる場合があり、ユーザはシステムのセットアップ時にこのことを考慮しておく必要があります。



この図は、STM32MP1 マイクロプロセッサでのトレースおよびデバッグ環境を表しており、ターゲット側ではコアやハードウェアペリフェラルとインタフェース接続するそれぞれのハードウェアパスおよびソフトウェアパスを示しています。
ホスト側では、ユーザとインタフェース接続して、トレースおよびデバッグのデータを管理または表示できるそれぞれのツールグループを表しています。
ここでは、これが一般的なトレースおよびデバッグのセットアップ図と言えます。すべてのボードに存在するとは限らない外部ターゲットインタフェースもあります。たとえば、トレースおよび I/O プローブインタフェースはディスカバリボードでは存在しません。



この図は、ハードウェアおよびソフトウェアのブロックを表しており、STM32MP1 リファレンスマニュアルにある詳細なデバッグブロック図と並行して使用できます。ここでは、STM32MP1 チップに組み込まれている主な Arm CoreSight ブロック IP をハイライトしています。

組込みカーネルの観察(1/3)

- カーネルの観察がその他のデバッグメソッドを使用する前の第一歩
- Linux カーネルはバッファにその動作を記録
  - 「printk」は printf と同じく、Linux カーネルのログバッファにメッセージを格納するために使用
  - 「dmesg」は、コンソールにログバッファの内容(異なるレベルの情報)を出力して、カーネル動作に関する情報を取得するために使用するコマンド
- Linux はファイル指向のオペレーティングシステムです。
  - 現在の設定、実行中のプロセスやアプリなどに関する多くの情報がファイルとして格納されている
    - これらのファイルは、/dev、/proc、/sys、/var のディレクトリに
  - /sys/kernel/debug(*debugfs* という名前)は、おそらく最も重要なファイルの 1 つ
- ペリフェラルレジスタは容易に出力可能
  - 出力されたレジスタは、ハードウェアの「ソフトウェア」を表す

それでは、UART 設定を中心とした例で示しながら、基本的な組込みカーネルを観察してみましょう。
カーネルの観察はその他のデバッグメソッドを使用する前の第一歩です。
Linux カーネルは、コアコンポーネントやドライバからの情報を格納するためにサーキュラログバッファを使用しています。

printk はこのバッファにいくつかのメッセージを格納するために使用される基本的な機能です。このバッファの内容を読み出すシンプルな方法は、「dmesg」コマンドを使用することです。
すべてのログメッセージが異なるログレベル – 0～7 – に分類されることに注意してください。この分類をもとに、読出し時にこれらのメッセージをフィルタできます。

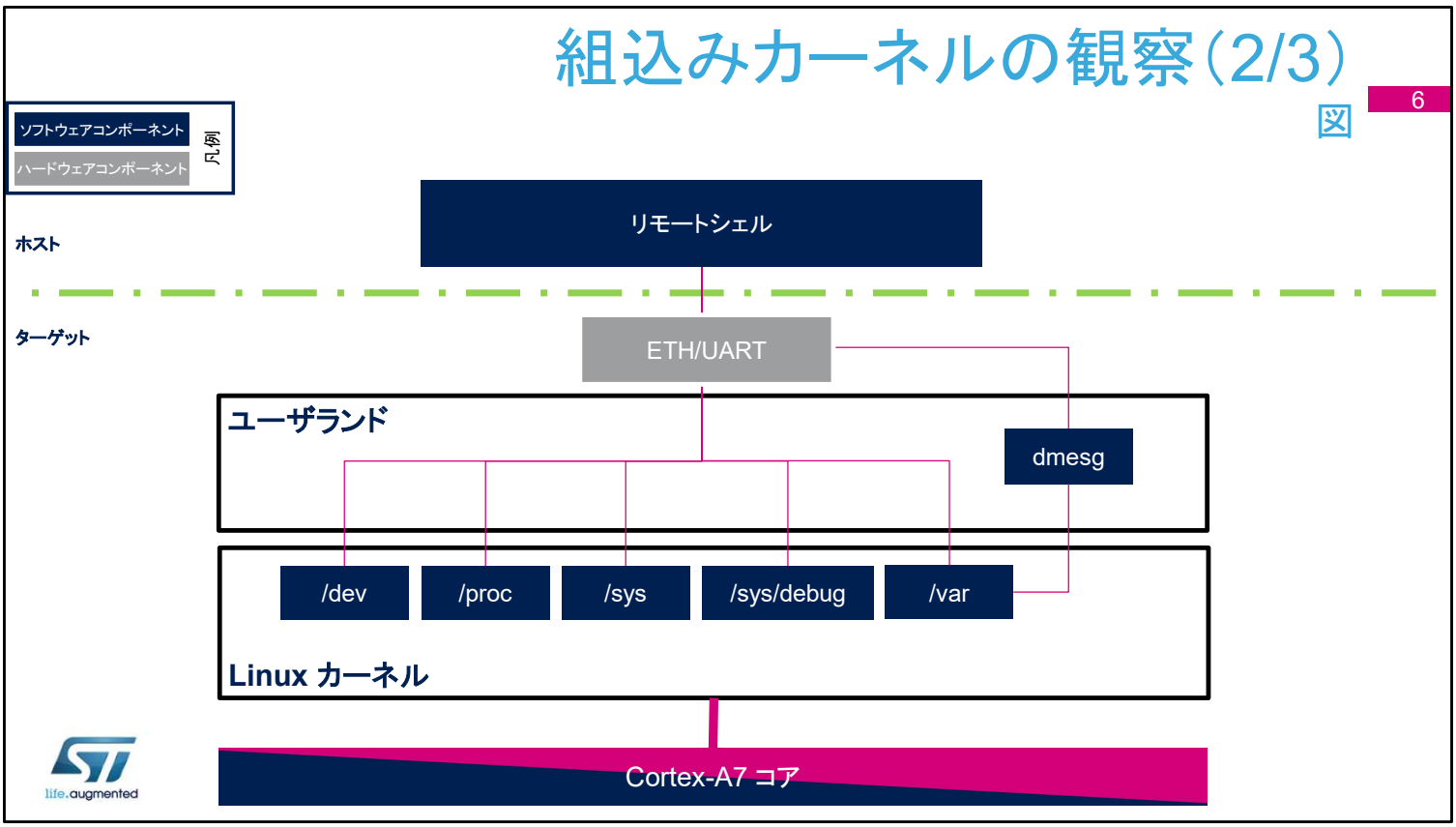
Linux カーネルはファイルシステム指向のオペレーティングシステムであることを覚えておいてください。これは、現在の設定、実行中のプロセスやアプリケーションに関するすべての情報がファイルとして格納され、トレースおよびデバッグツールでアクセスして読み出せることを意味します。このファイルシステムでは、STM32MP1 ハードウェアレジスタ設定に関するステータスを取得できます。これはシステム設定を管理してシステム動作を監視するために重要となります。

debugfs も、さらに多くのデバッグ情報を提供するファイルシステムの重要点です。このトレーニングで使用例をいくつか紹介します。

組込みカーネルの観察(2/3)

6

図



この図は、トレースおよびデバッグのアーキテクチャ図の抜粋です。ここでは、Linux カーネル観察のパスに絞っており、前のスライドに示した情報を要約しています。

組込みカーネルの観察(3/3)

実践 - デバッグシナリオ – UART 問題

- このプレゼンテーションのこれまでのスライドに示した組込みカーネルの観察を実践するために、wiki の記事が参照可能
  - [Trace_and_debug_scenario_-_UART_issue#Embedded_kernel_observations]
- ハードウェアデバイス上の問題をデバッグするための前提条件は次の通り
  - ピン配置のハードウェアボード図と、ピン設定のオルタネート機能の { STM32MP1 データシート }
  - 関連レジスタの説明に関する { STM32MP1 リファレンスマニュアル }

life.augmented

このプレゼンテーションのこれまでのスライドに示した組込みカーネルの観察を実践するために、wiki の記事を参照できます。
[Trace_and_debug_scenario_-_UART_issue#Embedded_kernel_observations]

前提条件として、リファレンスマニュアル、STM32MP1 チップのデータシート、ボードのハードウェア図を入手することをお勧めします。
上記の関連資料で次のことが可能になります。

- 各関連レジスタに関する説明全文と設定情報の把握
- 正確なハードウェア設定の把握

トレースおよびデバッグのツール（1/3）

8

- Linux コミュニティで提供されているトレースおよびデバッグのツールは膨大
  - wiki ページにある最も使用する役立つ主なツール：[Linux_tracing,_monitoring_and_debugging]
- OpenSTLinux の公式イメージに、役立つツールのほとんどが組み込まれている
  - strace、ip、ethtools、perf、netdata など
- Linux カーネルに組み込まれた 2 つの主なトレースおよびデバッグ機能
  - トレースするカーネルの一部を動的に選択する動的デバッグ（dyndbg）
    - 必要に応じて動的にソースコードにあるトレースメッセージを表示
    - dyndbg の情報は /sys/kernel/debug/dynamic_debug に格納
    - wiki ページの[How_to_use_the_kernel_dynamic_debug] を参照
  - カーネルの深いトレース用の Linux カーネルトレース機能
    - コアトレースフレームワークで、実行タイミングを含むイベントや関数呼び出しを追跡可能
    - ftrace などのツールでカーネルトレース機能から情報を収集
    - OpenSTLinux Weston イメージではトレース機能はデフォルトでセットされていません（**カーネルを再コンパイルする必要がある**）。

life.augmented

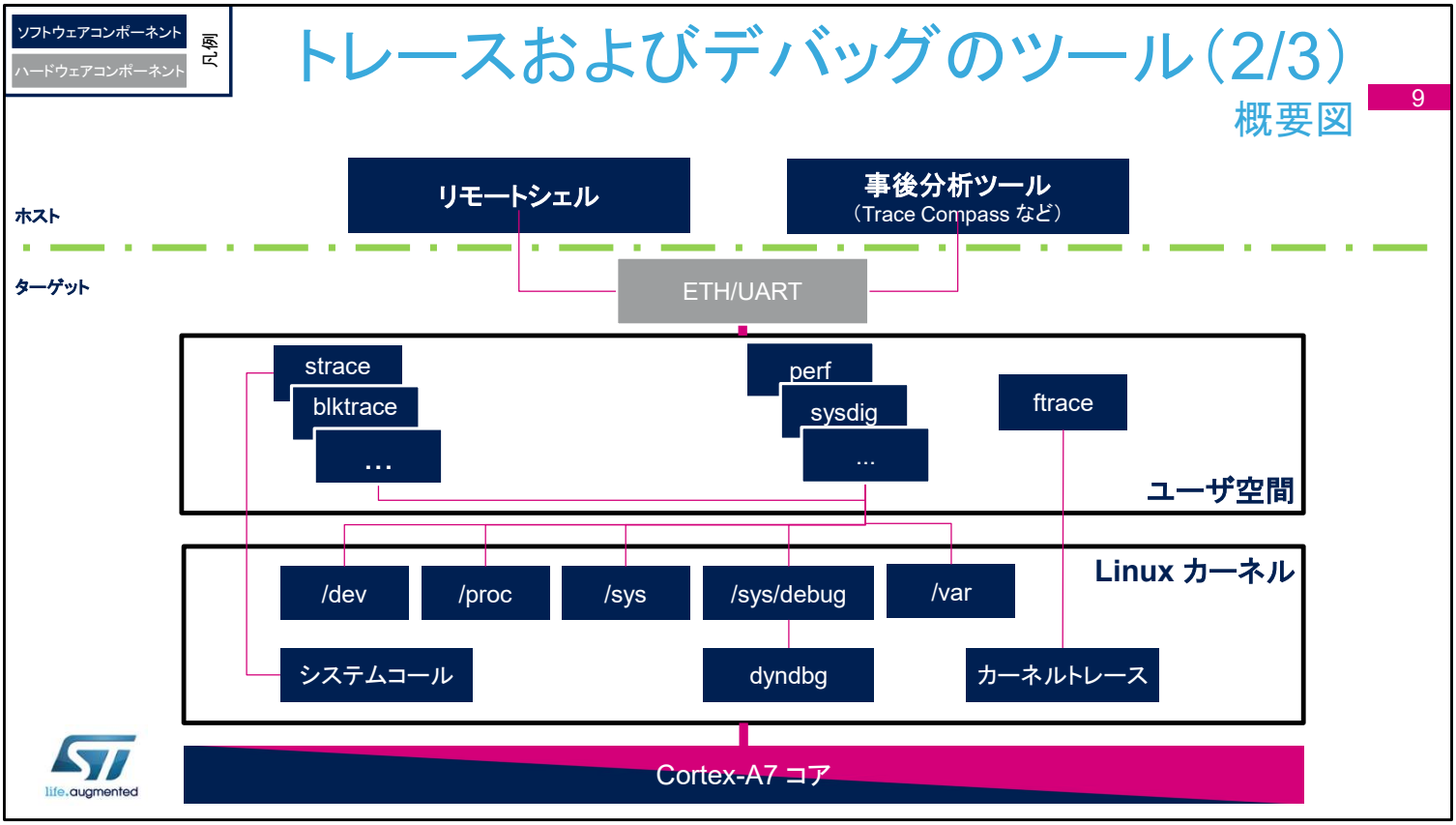
ここでは、STM32MP1 マイクロプロセッサで使用できるソリューションに関する概要の紹介や、引き続き UART 問題の例を示しつつ、トレースおよびデバッグのツールの情報を説明します。
Linux カーネルには、より深くカーネルを観察するためのツール一式が組み込まれています。

Linux コミュニティでも、主にユーザ空間レベルでのツールを提供しています。これらはファイルシステム、特に debugfs から複数の情報を取得できます。

Linux のトレース、監視、デバッグに関する wiki ページは、STM32MP1 に移植した Linux カーネルのトレースおよびデバッグのソリューションに関する優れたエントリポイントとなります。

トレースおよび監視のセットアップを始めるにあたり、主な 3 つの軸が提示されています。

- OpenSTLinux ディストリビューションにデフォルトで存在するツールには、役立つ情報を提供するものもあります。これらのツールの詳細については、対応する wiki の記事に記載されています。
- ここでは、OpenSTLinux に存在する Linux カーネルに組み込まれた 2 つの主なトレースおよびデバッグ機能についてハイライトします。
  - 動的デバッグは重要なトレース機能です。トレースバッファに負荷がかからないように、必要に応じてトレースを開始します。これらのトレースは特定の構文でソースコードに存在します（dev_dbg または pr_debug）。対応する wiki の記事を参照できます。
  - Linux カーネルのトレース機能は、強力なトレース環境でもあり、Linux カーネルのコード実行に関する情報を提供します。関数呼び出しをトレースするために使用され、パフォーマンスを分析します。このトレース機能はコード実行において侵入型であるため、デフォルトでは有効になっていないことに注意してください。すべての詳細は ftrace ツールの wiki の記事に記載されています。



この図は、ユーザ空間レベルで使用できる一部のツールを含んだ Linux カーネルのトレースおよび監視に関するリンクを表したブロック図です。

トレースおよびデバッグのツール（3/3）

実践 - デバッグシナリオ – UART 問題

- このプレゼンテーションのこれまでのスライドに示したトレースおよびデバッグのツール情報を実践するために、wiki の記事を参照可能
  - [Trace_and_debug_scenario_-_UART_issue#Trace_and_debug_tools]
- これらのツールを使用するための前提条件は次の通り
  - 新しい設定で Linux カーネルを再コンパイルする方法の把握
  - 特に関連するドライバのソースコードなどカーネルのソースコードの入手

life.augmented

このプレゼンテーションのこれまでのスライドに示したトレースおよびデバッグのツール情報を実践するために、wiki の記事を参照できます。[Trace_and_debug_scenario_-_UART_issue#Trace_and_debug_tools]

次の前提条件を整えることをお勧めします。

- カーネル設定オプションを変更してカーネルイメージを再コンパイルできる STM32MPU 組込みソフトウェアパッケージ（開発者パッケージまたはディストリビューションパッケージ）
- 指定のドライバに使用可能なデバッグトレースを確認できる Linux カーネルのソースコード

高機能なトレースおよびデバッグのツール(1/3)

概要詳細

- ハードウェアトレースの利用
  - ハードウェアトレースは、ハードウェアデバッグポート(HDP)または JTAG/トレースコネクタを使用するローレベルのトレースです。これはボード設定に依存することに注意
  - ハードウェアプローブ(ST-Link、U-Link、スコープまたはロジックアナライザ)が必要
- ソースコードで直接アクティブなデバッグを実行
  - ステップバイステップ実行というブレークポイントで実行してソースコードを制御
  - gdb はアクティブなデバッグを実行する Linux ツール

では、ハードウェアトレースを利用する高機能なトレースおよびデバッグのツールと、コアでのコード実行時に直接実行されるアクティブなデバッグを見てみましょう。
使用している STM32MP1 ボードに応じて、トレースや信号を取得するためのハードウェアデバッグポートへの完全アクセスまたは部分アクセスが可能です。

次の 3 種類の主なハードウェアデバッグポートが提供されており、ボードで使用できます。

- ST-Link や JTAG コネクタなどのハードウェアプローブを使用した JTAG およびシリアルワイヤポート(SWD)は、ソフトウェア実行およびメモリアクセス(読出しおよび書込み)に役立ちます。
- Arm CoreSight 信号を供給できるハードウェアトレースコネクタも、コアでのソフトウェア実行に紐付いています。
- ハードウェアデバッグポート(HDP)は、内部信号を観察できます。これは Linux カーネルドライバによって設定され、出力信号を GPIO に配置する内部ペリフェラルです。

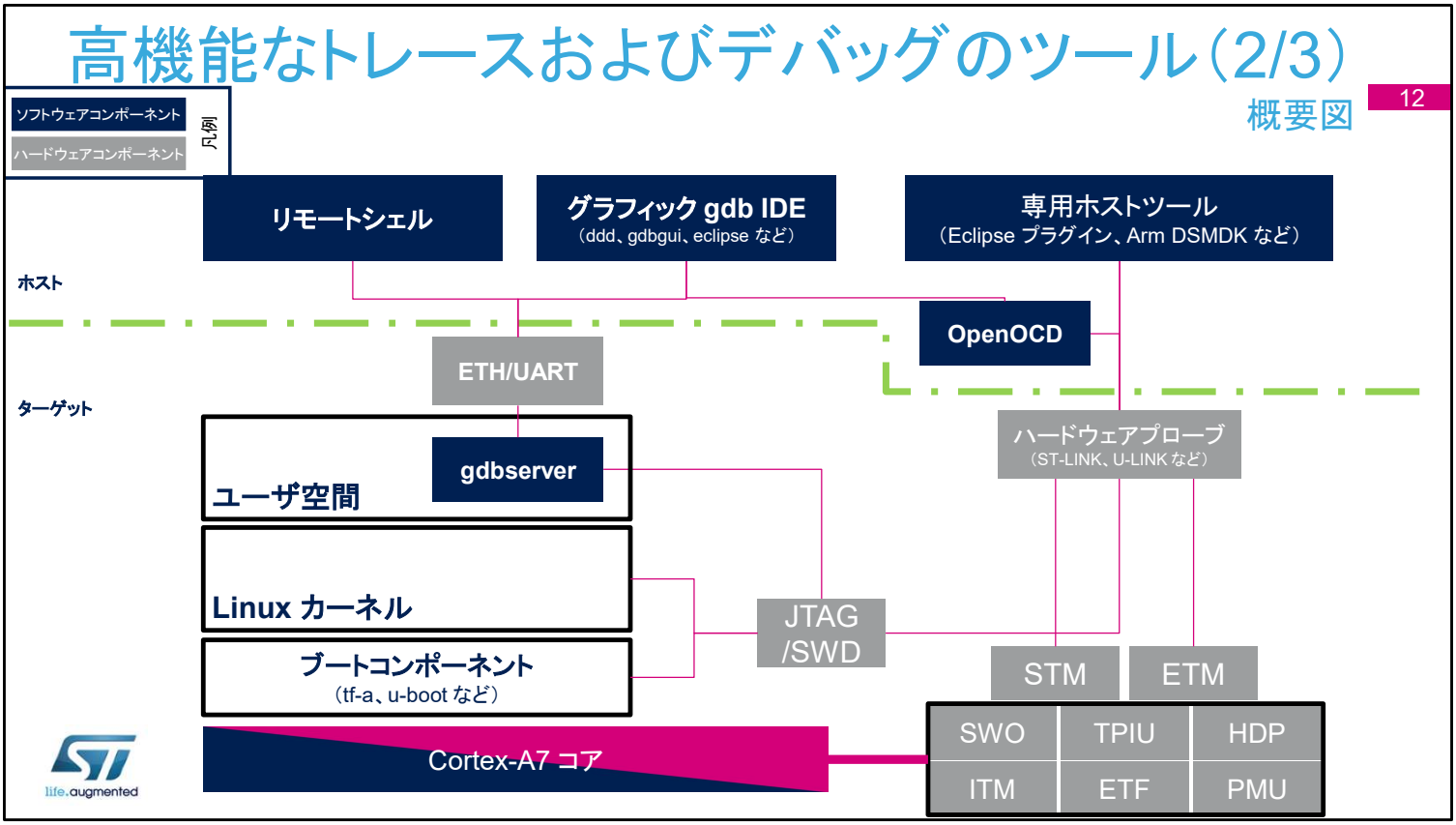
このデバッグポートの重要な用途として、実行中のソースコードで直接アクティブなデバッグが行えます。
ブレークポイントの追加、ステップバイステップ実行、変数のトレースによってソフトウェア実行を管理できます。

この環境では、ST-Link プローブを使用してホスト PC にターゲットを接続し、JTAG/SWD ポートを制御できる OpenOCD インタフェースにリンクされた GDB デバッガインタフェースをデバッグツールとして使用します。
詳細については、[GDB] の wiki ページを参照してください。

高機能なトレースおよびデバッグのツール（2/3）

12

概要図



ここでは、前のスライドで説明したハードウェアデバッグ設定について表しています。

高機能なトレースおよびデバッグのツール（3/3）

実践 - デバッグシナリオ – UART 問題

- このプレゼンテーションのこれまでのスライドに示した高機能なトレースおよびデバッグのツール情報を実践するために、wiki の記事を参照可能
  - [Trace_and_debug_scenario_-_UART_issue#Control_execution_of_code_with_gdb]
- GDB を使用するための前提条件は次の通り
  - GDB/OpenOCD 環境向けの開発者パッケージ

life.augmented

このプレゼンテーションのこれまでのスライドに示した高機能なトレースおよびデバッグのツール情報を実践するために、wiki の記事を参照できます。[Trace_and_debug_scenario_-_UART_issue#Control_execution_of_code_with_gdb]

次の前提条件を整えることをお勧めします。

- GDB および OpenOCD 向けに使用できる STM32MPU 組込みソフトウェアの開発者パッケージ

役立つ wiki 関連ページ

- **https://wiki.st.com/stm32mpu**
- トレースおよびデバッグの環境概要
  - [STM32MP1_Platform_trace_and_debug_environment_overview]
- Linux 向けのトレースおよびデバッグのツール
  - [Linux_tracing,_monitoring_and_debugging]
- Linux カーネルファイルシステム
  - [File Hierarchy Standard (FHS)]
- 高機能なデバッグツール
  - [GDB]

life.augmented

STM32MP1 ユーザガイドには、複数の局面で開発者をサポートする役立つ wiki の記事が記載されています。

- プラットフォームの使用方法
- プラットフォームのトレース、監視、デバッグの方法

STM32MP1 のトレースおよびデバッグのソリューションに関する詳細は、wiki.st.com/stm32mpu にアクセスして、このプレゼンテーションで列挙された記事を検索してください。

ありがとうございました。