

libmetal および OpenAMP ユーザー ガイド

UG1186 (v2020.1) 2020 年 8 月 18 日

この資料は表記のバージョンの英語版を翻訳したもので、内容に相違が生じる場合には原文を優先します。資料によっては英語版の更新に対応していないものがあります。日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。

改訂履歴

次の表に、この文書の改訂履歴を示します。

セクション	改訂内容
2020 年 8 月 18 日 バージョン 2020.1	
<ul style="list-style-type: none">• 全般的な更新。• libmetal と PetaLinux ツールを使用した Linux デモ アプリケーションの有効化。	<ul style="list-style-type: none">• 全般的な更新。• コードの例を更新。

目次

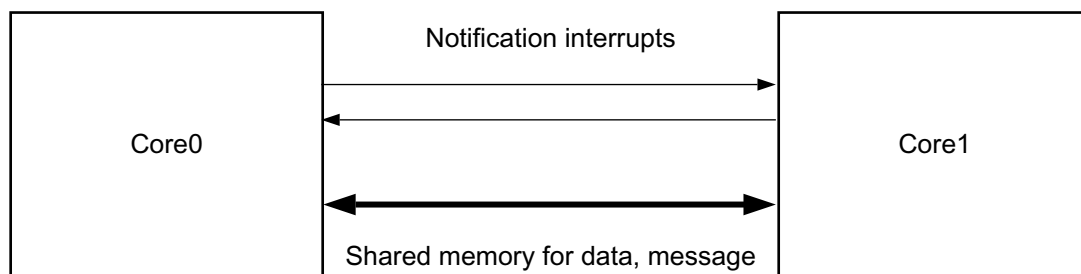
改訂履歴	2
第 1 章: 概要	
はじめに	4
第 2 章: libmetal	
概要	6
libmetal を使用したデバイスへのアクセス	7
ザイリンクス libmetal AMP デモ	11
第 3 章: OpenAMP	
概要	19
OpenAMP のコンポーネント	19
OpenAMP と libmetal 間の接続	21
簡単な OpenAMP アプリケーションの作成方法	21
OpenAMP デモ	25
第 4 章: システム設計に関する留意事項	
サポートされるコンフィギュレーション	45
その他の留意事項	46
既知の制限	46
付録 A: libmetal API	
libmetal API の関数	47
付録 B: OpenAMP API	
Remoteproc API	66
remoteproc API の関数	66
RPMsg の開発	72
RPMsg API の関数	72
付録 C: その他のリソースおよび法的通知	
ザイリンクス リソース	83
ソリューション センター	83
Documentation Navigator およびデザイン ハブ	83
ザイリンクス 資料	84
お読みください: 重要な法的通知	84

概要

はじめに

このユーザー ガイドでは、ザイリンクス Zynq® および Zynq UltraScale+™ MPSoC プラットフォーム上で複数プロセッサ間の通信機能を開発する方法を説明します。

開発上の基本的な概念は、互いにやり取りするプロセッサ間における割り込みと共有メモリという 2 つの動作原則に基づきます。



X19939-101217

図 1-1: プロセッサ間通信

libmetal ライブラリは、各種オペレーティング環境でデバイスへのアクセス、デバイス割り込みの処理、およびメモリの要求に使用する共通のユーザー API (アプリケーション プログラミング インターフェイス) を提供します。libmetal を使用して、ユーザー独自の AMP (非対称型マルチプロセッシング) ソリューションを構築できます。ザイリンクスは、デフォルトの AMP ソリューションとして OpenAMP (Open Asymmetric Multiprocessing) プロジェクトを使用します。OpenAMP は libmetal を基盤として構築され、リモート プロセッサ管理およびプロセッサ間通信用のフレームワークを提供します。この資料では、後続の各章で libmetal と OpenAMP の関係について説明します。

ソフトウェア ツールの要件

ここで説明する手順に従ってアプリケーションを構築するには、PetaLinux とザイリンクス Vitis™ が必要です。詳細は、「[ザイリンクス資料](#)」を参照してください。

- PetaLinux
- ザイリンクス Vitis

前提条件

OpenAMP フレームワークを効果的に使用するには、次の事項に関する基本的な理解が必要です。

- Linux、PetaLinux、およびザイリンクス Vitis
- JTAG ブートを使用してザイリンクス ボードをブートする方法
- Linux およびベアメタルで使用する remoteproc、RPMmsg、および virtIO コンポーネント

libmetal

概要

libmetal ライブラリは、OpenAMP オープンソース コミュニティによって管理されています。このライブラリは、各種オペレーティング環境でデバイスへのアクセス、デバイス割り込みの処理、およびメモリの要求に使用する共通のユーザー API を提供します。

libmetal は、次のオペレーティング システム/ソフトウェア構成で利用可能です。

- Linux (カーネル内の UIO および VFIO のサポートに基づく Linux ユーザー空間)
- FreeRTOS
- ベアメタル環境

次のアーキテクチャ図は、ユーザー アプリケーションがどのように libmetal ライブラリにアクセスするかを示しています。



図 2-1: libmetal アーキテクチャ

libmetal API の詳細は、libmetal のソース コード [\[参照 6\]](#) を参照してください。

libmetal を使用したデバイスへのアクセス

libmetal を使用することで、ユーザーは各種のオペレーティング環境と同じ方法でデバイスにアクセスできます。

libmetal を使用する手順は次のとおりです。

1. libmetal 環境を起動します。
2. デバイスを追加します。
3. デバイスを開きます。
4. 必要に応じて割り込みを登録します。
5. libmetal API を使用して、デバイス レジスタの書き込みと読み出しを実行します。
6. デバイスを閉じます。
7. libmetal 環境を終了します。

上記の手順については、次の各セクションで説明します。

プラットフォームによって、デバイスの抽象化は異なります。次の表は、libmetal がどのようにデバイスを管理するかをプラットフォームごとに示しています。

表 2-1: libmetal デバイス

Linux	ベアメタルおよび FreeRTOS
1. デバイスはデバイス ツリー内で記述されます。	1. デバイス ツリーの抽象化がないため、デバイスを開く前に、デバイスが静的に定義されている必要があります。
2. 「プラットフォーム」バスの定義は Linux カーネル内にあります。Linux は、この定義を使用してメモリ マップド デバイスを表現します。	2. バスの抽象化の規格はありません。libmetal ライブラリはデバイス管理用の汎用バス構造を定義します。

libmetal 環境の起動、デバイスを追加して開く

1. metal_init() を呼び出して、libmetal 環境を初期化します。

```
struct metal_init_params metal_param = METAL_INIT_DEFAULTS;
metal_init(&metal_param);
```
2. デバイスを追加します。
 - a. この手順は、ベアメタルまたは FreeRTOS の場合にのみ必要です。これは、ベアメタルではデバイスを記述するデバイス ツリーなどの規格が使用されないためです。
 - b. libmetal デバイスを静的に定義し、適切なバスに登録します。
 - c. 次のコード スニペットは、ベアメタルまたは FreeRTOS 用にどのようにトリプル タイマー カウンター デバイスを静的に定義するかを示しています。

- d. `metal_device` 構造体を初期化する際は、名前の文字列、デバイス用のバス、領域の数、デバイス内の各領域のテーブル、該当するバスのデバイスを追跡するノード、デバイスあたりの IRQ の数、および IRQ ID (必要な場合) を指定します。

```
const metal_phys_addr ipi_phy_addr = 0xff310000;
static struct metal_device static_dev = {
    .name = "ff310000.ipi",
    .bus = NULL, /* will be set later in metal_device_open() */
    .num_regions = 1, /* number of I/O regions */
    .regions = {
        {
            .virt = (void *) 0xff310000, /* virtual address */
            .physmap = &ipi_phy_addr, /* pointer to base physical address of the I/O region */
            .size = 0x1000, /* size of the region */
            .page_shift = (-1UL), /* page shift. In baremetal/FreeRTOS, memory is flat, no pages */
            .page_mask = (-1UL), /* page mask */
            .mem_flags = DEVICE_NONSHARED | PRIV_RW_USER_RW, /* memory attributes */
            .ops = {NULL}, /* no user specific I/O region operations. If don't want to use the default ones, you can define yours. */
        }
    },
    .node = {NULL}, /* will be set by libmetal later. used to keep track of the devices list */
    .irq_num = 1, /* number of interrupts of this device */
    .irq_info = (void *)65, /* interrupt information, here is the irq vector id */
};
metal_register_generic_device(static_dev);
```

Linux ユーザー空間の `libmetal` では、デバイスはデバイス ツリー内に配置されている必要があります。次に例を示します。

```
amba {
    ipi_amp: ipi@ff340000 {
        compatible = "ipi_uio"; /* used just as a label as libmetal will bind this device as UIO device */
        reg = <0x0 0xff340000 0x0 0x1000>;
        interrupt-parent = <&gic>;
        interrupts = <0 29 4>;
    };
};
```

3. デバイスを開きます。

デバイスを開いたらメモリ マップド デバイスの I/O 領域にアクセスし、必要に応じて割り込みを取得します。

```
struct metal_device *dev;
... // instantiate device here
metal_device_open(BUS_NAME, DEVICE_NAME, &dev);
```


割り込みの登録、デバイスレジスタの書き込みおよび読み出し

このセクションでは、既に libmetal 環境とレジスタ デバイス (必要な場合) を初期化し、これらのデバイスを開いていることを前提としています。

ベアメタルまたは FreeRTOS では、たとえば libmetal などの IPI (プロセッサ間割り込み) と共有メモリを使用する GIC (汎用割り込みコントローラー) を明示的に初期化する必要があります。

注記: 後続のセクションで、『Zynq UltraScale+ MPSoC テクニカル リファレンス マニュアル』(UG1085) [参照 2] の第 13 章で説明した Zynq UltraScale+ MPSoC ハードウェアの IPI エlement について説明します。

デバイスを閉じ、libmetal 環境を終了

libmetal API を使用してデバイスにアクセスした後、次のようにデバイスを閉じ、libmetal 環境を終了します。

```
/* Close the opened device */
metal_device_close(device);
/* Close the libmetal environment */
metal_finish();
```

libmetal を使用した IPI と共有メモリへのアクセス

Zynq UltraScale+ MPSoC の IPI ハードウェア

IPI (プロセッサ間割り込み) は、プロセッサ間の通知メッセージに使用できます。次の例では、IPI 共有バッファを使用しません。libmetal は IPI ドライバーを備えていません。libmetal はデバイスとしての IPI とやり取りする方法を提供するだけであり、ユーザーが IPI を管理する必要があります。

libmetal ユーザーは、libmetal ライブラリを使用して IPI にジェネリック デバイスとしてアクセスできます。IPI にアクセスする方法は、アプリケーションで定義する必要があります。スタンドアロン IPI ドライバーを使用すると、IPI ブロック間のメッセージの送受信方法はドライバによって定義されます。

注記: Linux ユーザー空間内の libmetal は、IPI バッファの使用を許可しません。IPI バッファは、PMU ファームウェアとのやり取り専用で使用され、Arm® Trusted Firmware (ATF) からしかアクセスできません。

metal_io_read32() および metal_io_write32() を使用して、IPI レジスタとやり取りできます。また、libmetal IRQ API を使用して、IPI 割り込みを処理できます。

Zynq® UltraScale+™ MPSoC の IPI レジスタにアクセスし、IPI 割り込みを処理するコードの例を次に示します。

この例では、ベアメタル/FreeRTOS 用に IPI libmetal デバイスを静的に定義しています。

```
static struct metal_device ipi_dev = { /* IPI device */
    .name = "ff310000.ipi", /* device name */
    .bus = NULL, /* bus. This field is NULL as it does not need to be set. It will be
set in metal_device_open() */
    .num_regions = 1, /* number of I/O regions in device */
    .regions = { /* define structure of each I/O region in device */
        {
            .virt = (void*) 0xFF3100, /* virtual address */
            .physmap = 0xFF3100, /* physical address */
            .size = 0x1000, /* size of region */
            .page_shift = (sizeof(metal_phys_addr_t) << 3), /* page shift */
            .page_mask = (unsigned long) (-1), /* page mask */
        }
    }
};
```

```

        .mem_flags = DEVICE_NONSHARED | PRIV_RW_USER_RW, /* memory flags */
        .ops = {NULL}, /* user-defined memory operations. We do not have any custom
operations so leave this as NULL.*/
    },
    .node = {NULL}, /* node to point to device in list of nodes on bus. This will be
set in metal_device_open, so leave as NULL */
    .irq_num = 1, /* The number of IRQs per device. In this case we are using only one
interrupt. */
    .irq_info = (void*) 65, /* IRQ ID*/
};
* Open the IPI device, use the IPI device as follows:
/* open the IPI device */
metal_device_open("generic", "ff310000.ipi", &ipi);
/* Get the IPI device libmetal I/O region */
io_region = metal_device_io_region(ipi, 0);
/* disable IPI interrupt */
metal_io_write32(ipi_io_region, IPI_IDR_OFFSET, IPI_MASK);
/* clear old IPI interrupt */
metal_io_write32(ipi_io_region, IPI_ISR_OFFSET, IPI_MASK);
/* Register IPI irq handler */
metal_irq_register(ipi_irq, ipi_irq_handler, ipi_dev, private_data);
/* Enable IPI interrupt */
metal_io_write32(ipi_io_region, IPI_IER_OFFSET, IPI_MASK);

```

共有メモリ

libmetal は、メモリ デバイスにアクセスしてデバイスとやり取りする方法を提供します。ただし、メモリのタイプはユーザーが定義する必要があります。

Linux ユーザー空間では、libmetal は UIO (Userspace I/O) ドライバーを使用するため、メモリをデバイス メモリとして扱うやり取りのみに制限されます。

libmetal は、メモリ マップド I/O 領域および共有メモリ領域へのアクセスを可能にする、I/O 領域の抽象化を提供します。これには、順序の制約付きでメモリの読み出しおよび書き込みを実行するためのプリミティブや、仮想メモリをサポートするシステム上で物理アドレス指定と仮想アドレス指定の変換を行う機能が含まれます。

共有メモリ デバイスの静的な定義、オープン、読み出しおよび書き込みの例を次に示します。この例では、ベアメタル/FreeRTOS 用に共有メモリ libmetal デバイスを静的に定義しています。

```

static struct metal_device shm_dev = { /* shared memory device */
    .name = "3ed80000.shm", /* device name */
    .bus = NULL, /* device bus */
    .num_regions = 1, /* number of regions on device */
    {
        {
            .virt = (void*) 0x3ED80000, /* virtual address */
            .physmap = 0x3ED80000, /* physical address */
            .size = 0x800000, /* size of region */
            .page_shift = (sizeof(metal_phys_addr_t) << 3), /* page shift */
            .page_mask = (unsigned long)(-1), /* page mask */
            .mem_flags = NORM_SHARED_NCACHE | PRIV_RW_USER_RW, /* memory flags */
            .ops = {NULL}, /* user defined memory operations */
        }
    },
    .node = {NULL}, /* node to point to device in list of nodes on bus */
};

```

```

        .irq_num = 0, /* Number of IRQs per device. This is 0 because there are no
interrupts we want to use for this device.*/
        .irq_info = NULL, /* IRQ info. This is NULL because we are not using this device
for interrupts. */
    }
    * Open the shared memory device, use the shared memory device as follows:
    /* Open the shared memory device */
    ret = metal_device_open("platform", "3ed80000.shm", &dev); /* the first argument,
bus name, is 'platform' for generic platform. */
    /* get shared memory device IO region */
    io = metal_device_io_region(device, 0);
    /* read data from the shared memory*/
    metal_io_block_read(io, READ_OFFSET, destination, data_length);
    /* write data to the shared memory*/
    ret = metal_io_block_write(io, WRITE_OFFSET, source, data_length);

```

ザイリンクス libmetal AMP デモ

libmetal AMP デモンストレーション アプリケーションは、どのようにデバイス (すなわち、共有メモリおよび割り込み) を開いてアクセスするかを示します。

ザイリンクス Vitis および PetaLinux ツールには、libmetal ライブラリを使用して Zynq UltraScale+ MPSoC プラットフォーム上の APU (アプリケーションプロセッシングユニット) と RPU (リアルタイムプロセッサ) 間の簡単なプロセッサ間通信を構築する方法を示す libmetal デモが含まれています。

この例は、プロセッサ間通信に次のリソースを使用します。

- DDR (ダブルデータレート)
- 通知用の IPI (プロセッサ間割り込み)
- レイテンシとスループットの測定を示すトリプルタイマー カウンター

次のセクションでは、ザイリンクス Vitis および PetaLinux ツールを使用して libmetal サンプルを構築する方法を説明します。

libmetal AMP デモンストレーションには、次の要素が含まれます。

- 共有メモリ
- アトミック操作に対応する共有メモリ
- 共有メモリを使用する IPI
- IPI のレイテンシ測定
- 共有メモリのレイテンシ測定
- 共有メモリのスループット測定

ザイリンクス Vitis を使用した libmetal ベアメタル ファームウェアの構築

1. [Xilinx Vitis] ウィンドウから、[File] → [New] → [Application Projects] → [Next] をクリックして、アプリケーションプロジェクトを作成します。
 - a. ハードウェア (XSA) から新しいプラットフォームを作成します。
 - [Browse] をクリックして .xsa ファイルをインポートし、[Next] をクリックします。
 - b. アプリケーションプロジェクトの名前を入力します。
 - ターゲット プロセッサとして [psu_cortexr5_0] を選択し、[Next] をクリックします。
 - c. psu_cortex5_0 target の新しいスタンドアロン ドメインを作成し、[Next] をクリックします。
 - d. 利用可能なテンプレートから [Libmetal AMP Demo] を選択します。
 - e. [Finish] をクリックします。
2. アプリケーションを構築する前に、ザイリンクス Vitis の [Project Explorer] ビューから、生成されるアプリケーションのソースコードを確認します。libmetal デモンストレーション アプリケーションの主なソースファイルは次のとおりです。
 - sys_init.c: GIC の初期化などのシステム初期化と、IPI デバイスおよび共有メモリ用の libmetal デバイスの定義。
注記: psu_cortex_r5_1 を選択した場合は、sys_init.c で、IPI_BASE_ADDR を 0xFF320000 に変更し、IPI_IRQ_VECT_ID を 66 に変更します。
 - libmetal_amp_demod.c: プロセッサ間通信用に libmetal で IPI と共有メモリをどのように使用するかを示すデモ アプリケーション。
 - common.h: ザイリンクス libmetal AMP デモ内の複数のデモに必要な共有リソースおよび関数と各デモの関数ヘッダーを含む共通ファイル。
 - ipi_latency_demod.c: APU と RPU 間のレイテンシを測定するデモ アプリケーション。
 - ipi_shmem_demod.c: 共有メモリと IPI にどのようにアクセスするかを示す。
 - shmem_atomic_demod.c: アトミック操作を使用して共有メモリにどのようにアクセスするかを示す。
 - shmem_demod.c: APU と RPU 間の共有メモリの使い方を示す。
 - shmem_latency_demod.c: APU と RPU 間の共有メモリのレイテンシを測定するデモ アプリケーション。
 - shmem_throughput_demod.c: APU と RPU 間の共有メモリのスループットを測定するデモ アプリケーション。
3. アプリケーションプロジェクトを構築するには、Ctrl + B キーを押します。生成された ELF は、<rupu_app_proj>/Debug/ ディレクトリに置かれます。

libmetal と PetaLinux ツールを使用した Linux デモ アプリケーションの有効化

PetaLinux ツールを使用して、次の手順を実行します。

1. 適切なディレクトリに PetaLinux マスタープロジェクトを (パスにスペースを入れずに) 作成します。このガイドでは、プロジェクト名は `<plnx-proj-root>` です。

```
$ petalinux-create -t project -s <PATH_TO_PETALINUX_ZYNQMP_PROJECT_BSP>
```

注記: PetaLinux BSP は、

<https://japan.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools.html> から入手できます。

2. プロジェクト ディレクトリに移動します。

```
$ cd <plnx-proj-root>
```

3. 必要な rootfs パッケージとアプリケーションを有効にします。

```
$ petalinux-config -c rootfs
```

4. libmetal および sysfs パッケージが有効になっていることを確認します。

```
Filesystem Packages--->
  misc --->
    sysfsutils --->
      [*] libsysfs
  Libs --->
    libmetal--->
      [*] libmetal
```

5. libmetal デモ アプリケーションが有効になっていることを確認します。

```
Filesystem Packages --->
  libs --->
    libmetal-->
      [*] libmetal-demos
```

6. libmetal Linux デモンストレーション アプリケーション用のデバイス ツリーを設定します。

デバイス ツリーの変更は、`system-user.dtsi` に追加する必要があります。

Petalinux `system-user.dtsi` のパス:

```
<plnx-proj-root>/project-spec/meta-user/recipes-bsp/device-tree/files/
system-user.dtsi
```

注記: 予約済みのメモリ ノードは、共有メモリとファームウェア用です。ファームウェアをほかの場所にロードする場合は、このメモリ ノードを移動できます。`system-user.dtsi` ファイルにデバイス ツリー ノードを手動で追加する必要があります。

```
/{
  reserved-memory {
    #address-cells = <2>;
    #size-cells = <2>;
    ranges;
    rproc_0_reserved: rproc@0x3ed20000 {
      no-map;
      reg = <0x0 0x3ed20000 0x0 0x2000000>;
    };
  };
};
```

```

    amba {
        /* Shared memory */
        shm0: shm@0 {
            compatible = "shm_uio";
            reg = <0x0 0x3ed80000 0x0 0x1000000>;
        };

        /* IPI device */
        ipi_amp: ipi@ff340000 {
            compatible = "ipi_uio";
            reg = <0x0 0xff340000 0x0 0x1000>;
            interrupt-parent = <&gic>;
            interrupts = <0 29 4>;
        };
    };
};&ttc0 {
    compatible = "ttc0";
    status = "okay";
};

```

shm0 デバイス ツリー ノードは、libmetal アプリケーションによって、アドレス 0x3ed80000 から始まる共有メモリに使用されます。

remoteproc を使用してファームウェアをロードする場合は、デバイス ツリー内で remoteproc デバイス ノードを定義することもできます。

TCM と DDR の両方でメモリを使用するサンプル remoteproc デバイス ノードは、次のようになります。

注記: ファームウェア メモリは、ファームウェアのリンカー スクリプトに対応する必要があります。このアプリケーションのサンプル リンカー スクリプトは、次のウェブサイトに掲載されています。

https://github.com/OpenAMP/libmetal/blob/master/examples/system/generic/zynqmp_r5/zynqmp_amp_demo/lscript.ld

```

zynqmp-rpu {
    compatible = "xlnx,zynqmp-r5-remoteproc-1.0";
    #address-cells = <2>;
    #size-cells = <2>;
    ranges;
    core_conf = "split";

    r5_0: r5@0 {
        #address-cells = <2>;
        #size-cells = <2>;
        ranges;
        pnode-id = <0x7>;
        tcm_0_a: tcm_0@0 {
            reg = <0x0 0xFFFE00000 0x0 0x10000>;
            pnode-id = <0xf>;
        };
        tcm_0_b: tcm_0@1 {
            reg = <0x0 0xFFFE20000 0x0 0x10000>;
            pnode-id = <0x10>;
        };
    };
};

```

Linux 側の libmetal のサンプル ソース コードは、次のウェブサイトに掲載されています。

https://github.com/OpenAMP/libmetal/tree/master/examples/system/linux/zynqmp/zynqmp_amp_demo

- common.h
- ipi_latency_demo.c
- ipi_shmem_demo.c
- libmetal_amp_demo.c
- shmem_atomic_demo.c
- shmem_demo.c
- shmem_latency_demo.c
- shmem_throughput_demo.c
- sys_init.c
- sys_init.h

ザイリンクス Vitis 内での libmetal Linux デモの構築

PetaLinux は、meta-openamp を使用して libmetal ライブラリと libmetal Linux デモ アプリケーションを構築します。ユーザー独自の libmetal アプリケーションは、ザイリンクス Vitis を使用して作成できます。

ザイリンクス Vitis 内でアプリケーションを生成する手順を次に示します。

1. パッケージの sysroot を構築します。

```
$ petalinux-build -s
$ petalinux-package --sysroot
```
2. XVitis を実行します。
3. Linux アプリケーション用の新しいプラットフォーム プロジェクトを作成します。[File] → [New Platform Project] をクリックして、プラットフォーム プロジェクトの名前を指定します。

```
OS: Linux
Processor: psu_cortexa53
Linux sysroot: the sysroot you built from your PetaLinux project:
    "--sysroot=<plnx-proj-root>/images/linux/sdk/sysroots/aarch64-xilinx-linux
Click Next
```
4. ハードウェア (XSA) から新しいプラットフォームを作成します。[Browse] をクリックして、.xsa ファイルをインポートします。
 - オペレーティング システム (Linux) を選択します。
 - プロセッサ (psu_cortexa53) を選択します。
 - [Generate boot components] をオフにします。
 - [Using the boot components generated by the PetaLinux project] を選択して、[Finish] をクリックします。

5. Linux libmetal アプリケーションプロジェクトを作成します。[File] → [New Application Project] → [Next] をクリックします。
 - 。 リポジトリからプラットフォームを選択して、前の手順で作成した Linux プラットフォームプロジェクトを選択します。アプリケーションプロジェクト名を指定し、プロセッサ (psu_cortexa53 SMP) を指定します。
 - 。 PetaLinux プロジェクト内で sysroots パスを指定し、次のディレクトリに移動します。
 - images/linux/sdk/sysroots/aarch64-xilinx-linux
 - 。 オプション:
 - RootFS - rootfs.tar.gz
 - カーネル イメージ - image.ub
 - 利用可能なテンプレート - Linux Empty Application
6.


```
C/C++ Build → Settings
      Tool Setting Tab Libraries
      Libraries (-l) add "metal"
```
7. https://github.com/OpenAMP/libmetal/tree/master/examples/system/linux/zynqmp/zynqmp_amp_demo に置かれているファイルを、アプリケーションの src ディレクトリにコピーします。
 - 。 common.h
 - 。 ipi_latency_demo.c
 - 。 ipi_shmem_demo.c
 - 。 shmem_atomic_demo.c
 - 。 shmem_demo.c
 - 。 shmem_latency_demo.c
 - 。 shmem_throughput_demo.c
 - 。 sys_init.c
 - 。 sys_init.h
 - 。 libmetal_amp_demo.c

注記: このデモは、デフォルトでは RPU 0 にアクセスします。デモが RPU 1 にアクセスするように変更する場合は、common.h 内の IPI マスク値を 0x200 (デフォルトの RPU1 IPI マスク) に変更します。

8. XViTis から構築された Linux アプリケーション実行ファイルとファームウェアを、次のように作成された Yocto レシピを使用して、PetaLinux ツールで構築された rootfs 内にインストールします。

```
petalinux-create -t apps -n <app_name> --enable
Modify the project-spec/meta-user/recipes-apps/<app_name>/<application name>.bb to
install the remote processor firmware in the RootFS as follows:
SUMMARY = "Simple test application"
SECTION = "PETALINUX/apps"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"
SRC_URI = "file://<linux-app> \
          file://<firmware> \
          "

S = "${WORKDIR}"
```



```

INSANE_SKIP_${PN} = "arch"

do_install() {
    # Install firmware into /lib/firmware on target
    install -d ${D}/lib/firmware
    install -m 0644 ${S}/<firmware> ${D}/lib/firmware/<firmware>

    # Install linux application into /usr/bin on target
    install -d ${D}/usr/bin
    install -m 0755 ${S}/<linux-app> ${D}/usr/bin/<linux-app>
}

FILES_${PN} = "/lib/firmware/<firmware> /usr/bin/<linux-app> "

```

注記: ZynqMP 上の libmetal Linux 側のデモが RPU1 にアクセスするには、
https://github.com/OpenAMP/libmetal/blob/master/examples/system/linux/zynqmp/zynqmp_amp_demo/common.h#L38 の
0x100 を 0x200 に変更します。

Linux デモ アプリケーションと Linux プロジェクトの構築

1. PetaLinux ツールプロジェクトに移動します。

```
$ cd <plnx_proj>
```

2. PetaLinux プロジェクトを構築します。

```
$ petalinux-build
```

カーネル イメージとデバイス ツリー バイナリは、<plnx-proj-root>/images/linux ディレクトリに置かれます。

ハードウェア上でのテスト

1. PetaLinux プロジェクトに移動します。

```
$ cd <plnx_proj>
```

2. PetaLinux プロジェクトを構築します。

```
$ petalinux-build
```

3. PetaLinux ブートを実行します。

```
$ petalinux-boot --jtag --kernel
```

問題が発生した場合は、これらのコマンドに -v を追加して、出力される文字を確認します。

4. remoteproc sysfs を使用して RPU ファームウェアをブートします。

ファームウェアは /lib/firmware ディレクトリに置かれている必要があります。

```

$ echo <firmware_name> > /sys/class/remoteproc/remoteproc0/firmware
$ echo start > /sys/class/remoteproc/remoteproc0/state

```

SD ブートなどのほかの手法を使用して、APU 上で Linux をブートし、RPU 上でファームウェアをブートすることも可能です。この例では JTAG ブートのみを扱います。

5. APU Linux ターゲット コンソール上で、XVitis を使用して構築した Linux アプリケーション上でデモ アプリケーションを実行するか、または PetaLinux BSP に付属の、あらかじめ構築された 「libmetal_amp_demo」 を使用します。このプロセスにより、次のような出力が生成されます。

```
# <linux libmetal application
metal: warning:    skipped page size 2097152 - invalid args
CLIENT> ***** libmetal demo: shared memory *****
metal: info:      meta
SERVER> Demo has started.

SERVER> Shared memory test finished

SERVER> ===== libmetal demo: atomic operation over shared memory =====

SERVER> Starting atomic add on shared memory demo.
l_uio_dev_open: No IRQ for device 3ed80000.shm.
CLIENT> Setting up shared memory demo.
CLIENT> Starting shared memory demo.
CLIENT> Sending message: Hello World - libmetal shared memory demo
CLIENT> Message Received: Hello World - libmetal shared memory demo
CLIENT> Shared memory demo: Passed.
CLIENT> ***** libmetal demo: atomic operation over shared memory *****
```

注記: XSDB は、アプリケーションのデバッグに使用できる手法の 1 つです。XSDB の使用方法の詳細は、『エンベデッド システム ツール リファレンス マニュアル』(UG1043) を参照してください。

OpenAMP

概要

OpenAMP (Open Asymmetric Multi-Processing) は、非対称型マルチプロセッシング (AMP) システム用ソフトウェアアプリケーションの開発に必要なソフトウェアコンポーネントを提供するフレームワークです。このフレームワークは、次の主な機能を提供します。

- リモート コンピューティング リソースおよび関連するソフトウェア コンテキストの管理に使用される、ライフ サイクル管理機能とプロセッサ間通信機能
- RTOS およびベアメタル ソフトウェア環境で利用可能なスタンドアロン ライブラリ
- Linux remoteproc、rmpmsg、および VirtIO といったアップストリーム コンポーネントとの互換性

OpenAMP のコンポーネント

RPMsg (リモート プロセッサ メッセージング)、VirtIO (仮想化モジュール) および remoteproc は、アップストリームの Linux カーネル内にインプリメントされます。OpenAMP ライブラリは、ベアメタル、FreeRTOS (リアルタイムオペレーティングシステム)、および Linux ユーザー空間の各環境向けに、これらコンポーネントのインプリメントをサポートします。

virtIO: OpenAMP ライブラリは、共有メモリ管理用の virtIO 規格をインプリメントします。virtIO は、ゲストデバイス上のドライバーだけがハイパーバイザーを使用して仮想環境内で動作していることを認識する、ネットワークおよびディスク デバイス ドライバー用の仮想化規格です。

remoteproc: remoteproc は、リモート プロセッサのライフ サイクル管理 (LCM) 機能を提供します。OpenAMP ライブラリが使用する remoteproc API は、Linux カーネル 3.18 およびそれ以降のインフラストラクチャに準拠しています。remoteproc は、リモート プロセッサのファームウェア リソース テーブルを介して公開される情報を使用して、システム リソースを割り当て、virtIO デバイスを作成します。remoteproc を使用して、OpenAMP ファームウェアに限らず、任意のファームウェアをロードできます。

RPMsg: この API は、AMP システム内の互いに独立したコア上で動作するソフトウェア間のプロセス間通信 (IPC) を可能にします。この API も、Linux カーネル 3.18 およびそれ以降の RPMsg バス インフラストラクチャに準拠しています。

次の図は、ザイリンクス Zynq および Zynq UltraScale+ MPSoC プラットフォーム内で OpenAMP がどのように使用されるかを示しています。

1. Linux カーネル マスターと RPU OpenAMP スレーブ

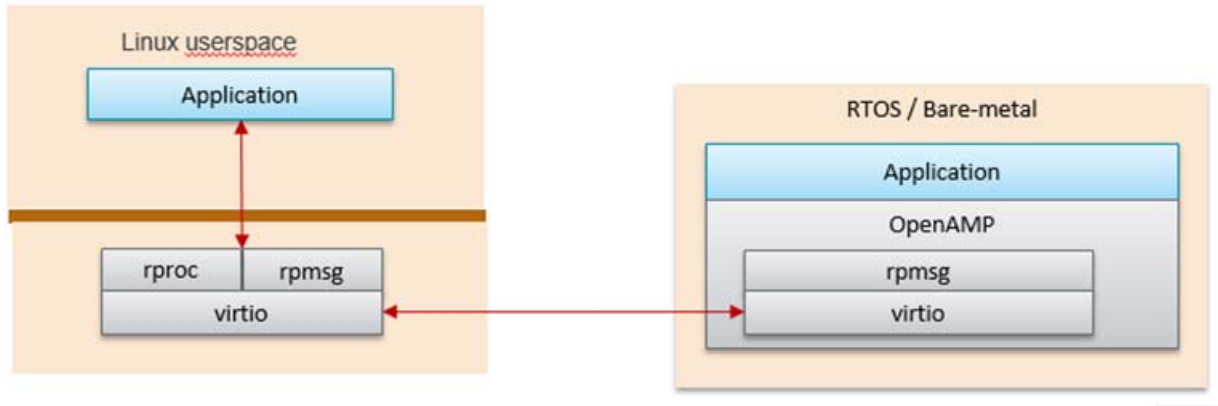


図 3-1: カーネル空間内の RPMsg インプリメンテーション

Linux カーネル空間は RPMsg および Remoteproc を提供しますが、RPU アプリケーションが Linux カーネル内の対応する RPMsg にアクセスするには、Linux がそのアプリケーションをロードする必要があります。これは Linux カーネルの RPMsg および Remoteproc インプリメンテーションの制限です。

2. Linux ユーザー空間の OpenAMP アプリケーションと RPU の OpenAMP アプリケーション

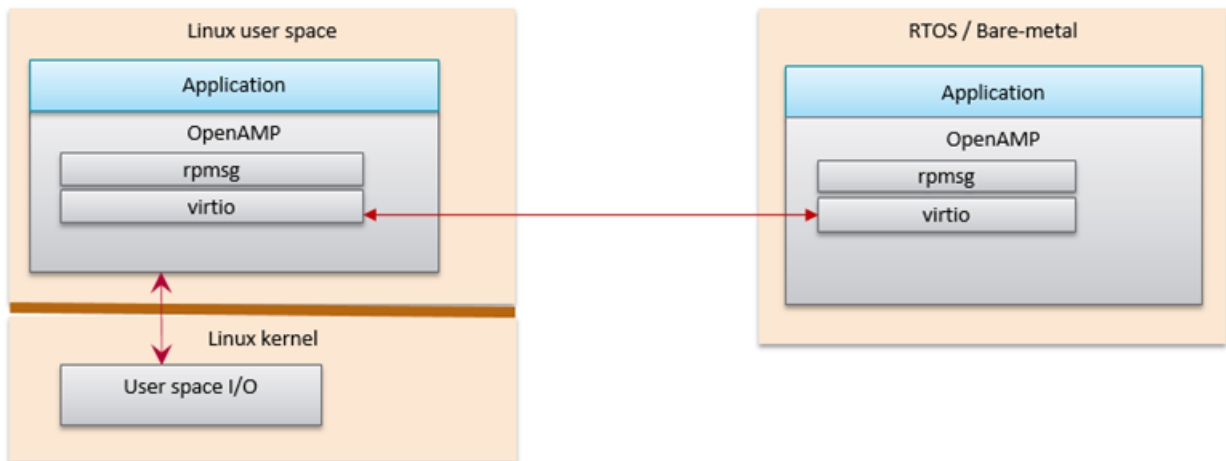


図 3-2: Linux ユーザー空間内の OpenAMP RPMsg インプリメンテーション

OpenAMP ライブラリは Linux ユーザー空間内でも使用できます。この構成では、リモート プロセッサは Linux ホスト プロセッサに対して独立して動作できます。

OpenAMP と libmetal 間の接続

OpenAMP と libmetal 間の接続について説明します。

OpenAMP は libmetal を抽象化層として使用し、デバイスへのアクセスと、割り込みおよび共有メモリの処理を実行します。libmetal はデバイスへのアクセスとメモリへのアクセスに共通のインターフェイスを提供することから使用されます。OpenAMP は libmetal を使用して IPI (プロセッサ間割り込み) と共有メモリにアクセスします。OpenAMP は、共有メモリ管理、ライフ サイクル管理、および通信の規格を利用します。libmetal と OpenAMP 間の接続を次の図に示します。

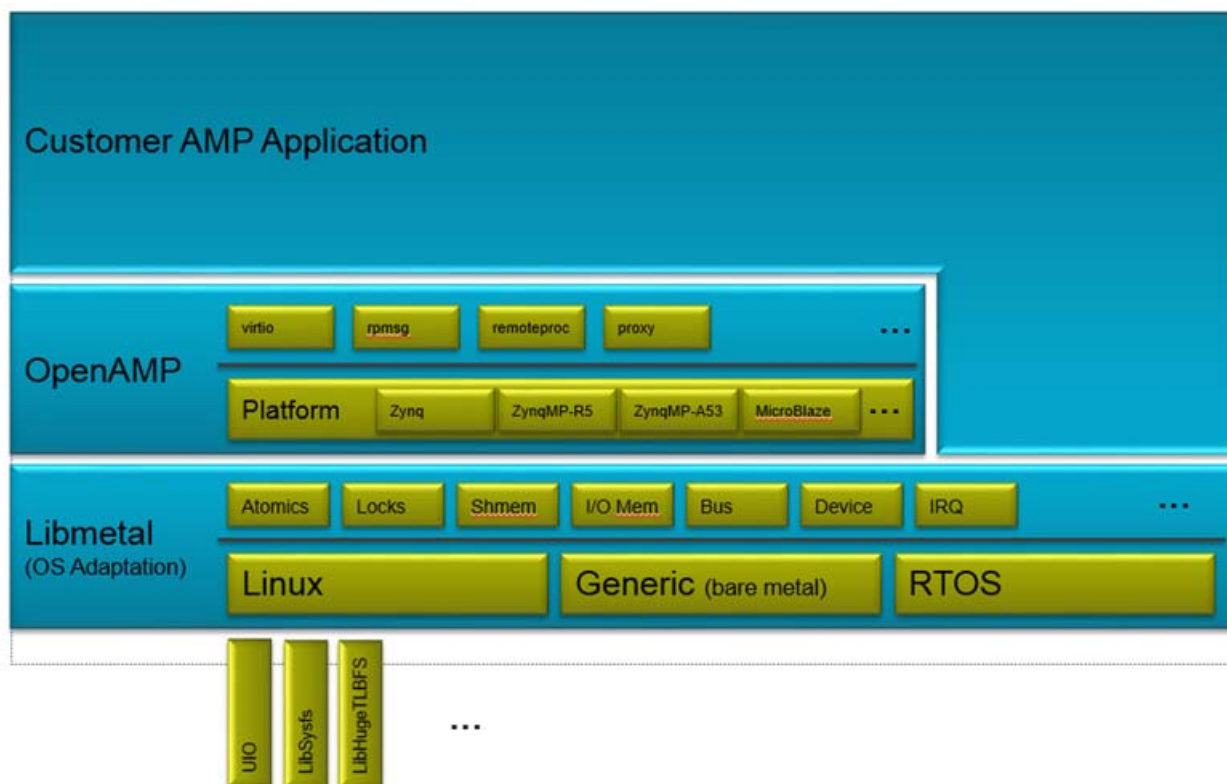


図 3-3: libmetal と OpenAMP の接続

簡単な OpenAMP アプリケーションの作成方法

OpenAMP アプリケーションを作成するには、次の手順に従う必要があります。

1. ファームウェア リソース テーブルを使用します。

このリソース テーブルは、OpenAMP アプリケーションに必要なファームウェア エントリを定義します。これはリモート `remote_proc` が必要とするシステム リソースのリストです。

2. リソース テーブルを使用して `remoteproc` 構造体を作成します。

3. RPMsg コールバック関数を定義します。
4. RPMsg virtio デバイスを作成します。
5. RPMsg エンドポイントを作成し、RPMsg デバイスとコールバック関数を関連付けます。
6. `rpmsg_send()` を使用して、リモート プロセッサにメッセージを送信します。
7. フレームワークを初期化した後の OpenAMP アプリケーションのフローは、RPMsg `send()` および I/O コールバック関数によるマスター プロセッサとリモート プロセッサ間の通信として機能する RPMsg チャンネルで構成されます。これを次のフロー図に示します。

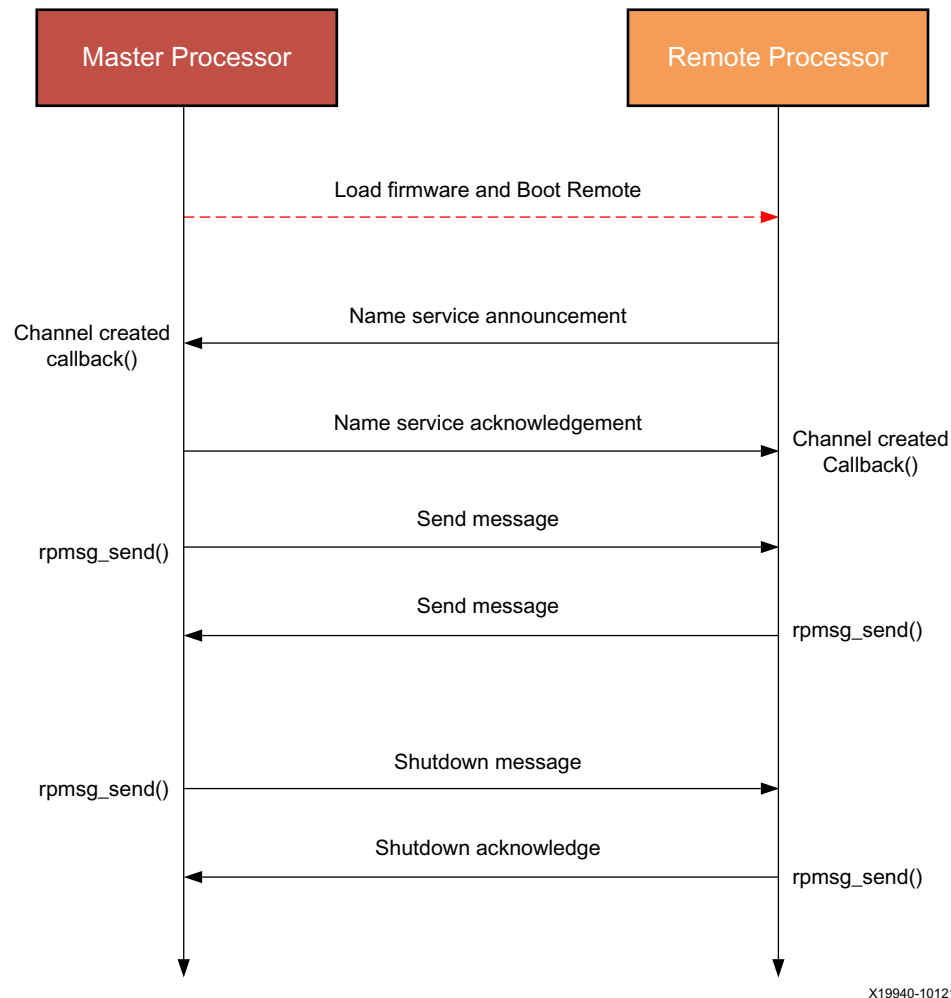


図 3-4: フロー図

リソース テーブル、`remoteproc` インスタンスおよび RPMsg コールバック関数を使用した、OpenAMP のセットアップとフローの例を次に示します。

```

struct resource_table table = {
/* Version number. If the structure changes in the future, this acts as
 * reference to what the structure is.
 */
.ver = 1,
/* Number of resources; Matches number of offsets in array */
.num = 2,

```

```

/* reserved (must be zero) */
.reserved = 0,
{ /* array of offsets pointing at various resource entries */
/* This RSC_RPROC_MEM entry set the shared memory address range. It is required to
tell the Linux kernel range of the shared memory the remote can access. */
*/
{RSC_RPROC_MEM, 0x3ed40000, 0x3ed40000, 0x100000, 0},
/* virtio device header */
{
RSC_VDEV, VIRTIO_ID_RPMSG_, 0, RPMSG_IPU_C0_FEATURES, 0, 0, 0,
NUM_VRINGS, {0, 0},
}
};
#include <openamp/remoteproc.h>
#include <openamp/rpmsg.h>
#include <openamp/rpmsg_virtio.h>

/* User defined remoteproc operations for communication */
struct remoteproc rproc_ops = {
.init = local_rproc_init;
.mmap = local_rproc_mmap;
.notify = local_rproc_notify;
.remove = local_rproc_remove;
};

/* Remoteproc instance. If you don't use Remoteproc VirtIO backend,
* you don't need to define the remoteproc instance.
*/
struct remoteproc rproc;

/* RPMsg VirtIO device instance. */
struct rpmsg_virtio_device rpmsg_vdev;

/* RPMsg device */
struct rpmsg_device *rpmsg_dev;

/* Resource Table. Resource table is used by remoteproc to describe
* the shared resources such as vdev(VirtIO device) and other shared memory.
* Resource table resources definition is in the remoteproc.h.
* Examples of the resource table can be found in the OpenAMP repo:
* - apps/machine/zynqmp/rsc_table.c
* - apps/machine/zynqmp_r5/rsc_table.c
* - apps/machine/zynq7/rsc_table.c
*/
void *rsc_table = &resource_table;

/* Size of the resource table */
int rsc_size = sizeof(resource_table);

/* Shared memory metal I/O region. It will be used by OpenAMP library
* to access the memory. You can have more than one shared memory regions
* in your application.
*/
struct metal_io_region *shm_io;

/* VirtIO device */
struct virtio_device *vdev;

```

```

/* RPMsg shared buffers pool */
struct rpmsg_virtio_shm_pool shpool;

/* Shared buffers */
void *shbuf;

/* RPMsg endpoint */
struct rpmsg_endpoint ept;

/* User defined RPMsg name service callback. This callback is called
 * when there is no registered RPMsg endpoint is found for this name
 * service. User can create RPMsg endpoint in this callback. */
void ns_bind_cb(struct rpmsg_device *rdev, const char *name, uint32_t dest);

/* User defined RPMsg endpoint received message callback */
void rpmsg_ept_cb(struct rpmsg_endpoint *ept, void *data, size_t len,
    uint32_t src, void *priv);

/* User defined RPMsg name service unbind request callback */
void ns_unbind_cb(struct rpmsg_device *rdev, const char *name, uint32_t dest);

void main(void)
{
    /* Instantiate remoteproc instance */
    remoteproc_init(&rproc, &rproc_ops);

    /* Mmap shared memories so that they can be used */
    remoteproc_mmap(&rproc, &physical_address, NULL, size,
        <memory_attributes>, &shm_io);

    /* Parse resource table to remoteproc */
    remoteproc_set_rsc_table(&rproc, rsc_table, rsc_size);

    /* Create VirtIO device from remoteproc.
     * VirtIO device master will initiate the VirtIO rings, and assign
     * shared buffers. If you running the application as VirtIO slave, you
     * set the role as VIRTIO_DEV_SLAVE.
     * If you don't use remoteproc, you will need to define your own VirtIO
     * device.
     */
    vdev = remoteproc_create_virtio(&rproc, 0, VIRTIO_DEV_MASTER, NULL);

    /* This step is only required if you are VirtIO device master.
     * Initialize the shared buffers pool.
     */
    shbuf = metal_io_phys_to_virt(shm_io, SHARED_BUF_PA);
    rpmsg_virtio_init_shm_pool(&shpool, shbuf, SHARED_BUFF_SIZE);

    /* Initialize RPMsg VirtIO device with the VirtIO device */
    /* If it is VirtIO device slave, it will not return until the master
     * side set the VirtIO device DRIVER OK status bit.
     */
    rpmsg_init_vdev(&rpmsg_vdev, vdev, ns_bind_cb, io, shm_io, &shpool);

    /* Get RPMsg device from RPMsg VirtIO device */
    rpmsg_dev = rpmsg_virtio_get_rpmsg_device(&rpmsg_vdev);

    /* Create RPMsg endpoint. */
    rpmsg_create_ept(&ept, rdev, RPMMSG_SERVICE_NAME, RPMMSG_ADDR_ANY,

```



```

        rpmsg_ept_cb, ns_unbind_cb);

/* If it is VirtIO device master, it sends the first message */
while (!is_rpmsg_ept_read(&ept)) {
    /* check if the endpoint has binded.
     * If not, wait for notification. If local endpoint hasn't
     * been bound with the remote endpoint, it will fail to
     * send the message to the remote.
     */
    /* If you prefer to use interrupt, you can wait for
     * interrupt here, and call the VirtIO notified function
     * in the interrupt handling task.
     */
    rproc_virtio_notified(vdev, RSC_NOTIFY_ID_ANY);
}
/* Send RPMsg */
rpmsg_send(&ept, data, size);

do {
    /* If you prefer to use interrupt, you can wait for
     * interrupt here, and call the VirtIO notified function
     * in the interrupt handling task.
     * If vdev is notified, the endpoint callback will be
     * called.
     */
    rproc_virtio_notified(vdev, RSC_NOTIFY_ID_ANY);
} while(!ns_unbind_cb_is_called && !user_decided_to_end_communication);

/* End of communication, destroy the endpoint */
rpmsg_destroy_ept(&ept);

rpmsg_deinit_vdev(&rpmsg_vdev);

remoteproc_remove_virtio(&rproc, vdev);

remoteproc_remove(&rproc);
}

```

OpenAMP デモ

このセクションでは、OpenAMP デモンストレーションの各アプリケーションについて説明します。

Linux マスターおよびベアメタル/FreeRTOS リモートのエコー テスト

このテスト アプリケーションは、マスターからリモートへ多数のペイロードを送信し、送信されたデータのインテグリティをテストします。

- エコー テスト アプリケーションは、remoteproc を使用して Linux マスターからリモート ベアメタル ファームウェアをブートします。
- 次に Linux マスターは、RPMsg を使用してリモート ファームウェアにペイロードを送信します。リモート ファームウェアは、RPMsg を使用して受信データをエコー バックします。
- Linux マスターはペイロードを検証し、出力します。

Linux マスターおよびベアメタル/FreeRTOS リモートの行列乗算

行列乗算アプリケーションは、マスター上に 2 つの行列を生成する、より複雑なテストを提供します。これらの行列はリモートに送信されます。リモートを使用して行列の乗算を実行します。次にリモートはマスターに結果を返信し、マスターが結果を表示します。

Linux マスターは、`remoteproc` を使用してベアメタル リモート ファームウェアをブートします。次に Linux マスターは、無作為に生成された 2 つの行列を `RPMMsg` を使用して送信します。

ベアメタル ファームウェアは 2 つの行列を乗算し、`RPMMsg` を使用してマスターに結果を返信します。

Linux マスターおよびベアメタル/FreeRTOS リモートのプロキシ アプリケーション

このアプリケーションは、Linux マスターとリモート コアの間にプロキシを作成します。これにより、リモート ファームウェアは、コンソールの使用とマスター上でのファイル I/O が可能になります。

Linux マスターは、`proxy_app` を使用してファームウェアをブートします。リモート ファームウェアは、マスター プロセッサ上の Linux ファイルシステム (FS) 上でファイル I/O を実行します。またリモート ファームウェアは、マスター コンソールを使用して入力とディスプレイ出力を受信します。

PetaLinux イメージのクイックトライ

次の基本的な手順を使用して Linux をブートし、あらかじめ作成されたイメージを使用して OpenAMP アプリケーションを実行します。ここでは、ZCU102 ボードを使用しています。

エコー テスト アプリケーションは、クワッドコア Cortex-A53 上で実行される Linux から FreeRTOS を実行しているシングル Cortex-R5 へパケットを送信し、Cortex-R5 はパケットを返信します。

1. あらかじめ作成された PetaLinux BSP tar ファイルから `BOOT.BIN`、`image.ub`、および `openamp.dtb` ファイルを SD カードに抽出します。OpenAMP 関連のデバイス ノードはデフォルトの `system.dtb` には含まれておらず、あらかじめ作成された `openamp.dtb` に含まれます。

```
host shell$ tar xvf xilinx-zcu102-v2019.2-final.bsp --strip-components=4 --wildcards
*/BOOT.BIN */image.ub */openamp.dtb
host shell$ cp BOOT.BIN image.ub openamp.dtb <your sd card>
```

注記: ボードに付属の BSP を使用して既に PetaLinux プロジェクトを作成している場合は、あらかじめ作成されたイメージが `<your project>/pre-built/linux/images/` ディレクトリに見つかります。

2. u-boot プロンプトに移動し、SD カードから Linux をブートします。

```
ZynqMP> mmcinfo && fatload mmc 0 0x10000000 image.ub && fatload mmc 0 0x14000000
openamp.dtb
Device: mmc@ff170000
Manufacturer ID: 3
OEM: 5344
Name: SL16G
Bus Speed: 100000000
Mode: UHS SDR50 (100MHz)
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 14.8 GiB
Bus Width: 4-bit
Erase Group Size: 512 Bytes
57212600 bytes read in 3718 ms (14.7 MiB/s)
40961 bytes read in 26 ms (1.5 MiB/s)
ZynqMP> bootm 0x10000000 0x10000000 0x14000000
```

注記: 上記の SD ブート手順の代わりに、ボードを JTAG ブートすることもできます。そのためには、JTAG ケーブルを接続して JTAG ドライバーをインストールし、付属の BSP を使用して PetaLinux プロジェクトを作成する必要があります。

これには、<your project>/pre-built/linux/images directory に移動し、system.dtb ファイルを openamp.dtb で置き換え、petalinux-boot --jtag --prebuilt 3 と入力する必要があります。

3. Linux ログインプロンプトにおいて、ユーザー名に root、パスワードに root を入力し、エコーテスト デモを実行します。

```
plnx_aarch64 login: root
Password:
root@plnx_aarch64:~# echo image_echo_test >
/sys/class/remoteproc/remoteproc0/firmware
root@plnx_aarch64:~# echo start > /sys/class/remoteproc/remoteproc0/state
[ 265.772355] remoteproc remoteproc0: powering up ff9a0100.zynqmp_r5_rproc
[ 265.779900] remoteproc remoteproc0: Booting fw image
echo_test_standalone_r5_0.elf, size 719860
[ 265.790005] zynqmp_r5_remoteproc ff9a0100.zynqmp_r5_rproc: RPU boot from TCM.
Starting application...
Initialize remoteproc successfully.
creating remoteproc virtio
initializing rpmsg shared buffer pool
initializing rpmsg vdev
initializing rpmsg vdev
Try to create rpmsg endpoint.
Successfully created rpmsg endpoint.
[ 265.797738] remoteproc remoteproc0: registered virtio0 (type 7)
[ 265.800388] virtio_rpmsg_bus virtio0: rpmsg host is online
[ 265.830254] remoteproc remoteproc0: remote processor ff9a0100.zynqmp_r5_rproc is
now up
[ 265.838381] virtio_rpmsg_bus virtio0: creating channel rpmsg-openamp-demo-channel
addr 0x0
root@xilinx-zcu102-2019_1:/lib/firmware# echo_test -d
virtio0.rpmsg-openamp-demo-channel.-1.0
```

Echo test start

Open rpmsg dev /dev/rpmsg0!

Echo Test Round 0

注記: この rpmsg デバイスドライバーは、アウト オブ ツリーの Linux カーネル モジュールです。起動時初期化スクリプトを作成しておけば、このドライバーをブート時にロードできます (『PetaLinux ツール資料: リファレンス ガイド』(UG1144) [参照 4] の例参照)。

RPU ファームウェア用 OpenAMP アプリケーションの構築

概要

ザイリンクス Vitis には、OpenAMP ベアメタル/FreeRTOS リモート アプリケーションの開発用テンプレートが含まれています。次の各セクションでは、ザイリンクス Vitis および PetaLinux ツールを使用して OpenAMP アプリケーションを作成する方法を説明します。

- ザイリンクス Vitis を使用して、ベアメタルまたは FreeRTOS リモート アプリケーションを作成します。

ザイリンクス Vitis 内でのリモート アプリケーションの構築

ザイリンクス Vitis を使用してリモート アプリケーションを構築する手順は次のとおりです。PetaLinux BSP には、リモート プロセッサ (Zynq® Cortex™-A9 #1 および Zynq UltraScale+™ MPSoC Cortex-R5 #0) 用にあらかじめ構築されたファームウェアが既に含まれています。この手順は、リモート プロセッサ上で動作するデモ アプリケーションを再構築する場合にのみ必要です。

OpenAMP 用アプリケーション プロジェクトの作成

- [Xilinx Vitis] ウィンドウから、[File] → [New] → [Application Projects] をクリックするか、または [Create Application Project] をクリックして、アプリケーション プロジェクトを作成します。
- ウィザードに従って、新しいアプリケーション プロジェクトの作成手順を実行します。
 - プラットフォームを選択するか、または Vivado でエクスポートした XSA からプラットフォーム プロジェクトを作成します。
 - システム プロジェクト内にアプリケーション プロジェクトを配置し、プロセッサと関連付けます。
 - アプリケーション ランタイムードメインを準備します。
 - アプリケーションのテンプレートを選択すると、直ちに開発を開始できます。
- BSP の OS プラットフォームを指定します。
 - ベアメタル アプリケーションの場合は [standalone]
 - FreeRTOS アプリケーションの場合は [freertos<version>_xilinx]
 - ハードウェア プラットフォームを指定します。
 - プロセッサを選択します。

Zynq UltraScale+ MPSoC デバイス (ZynqMP) では、Cortex-R5 (RPU) がサポートされています。

 - [psu_cortexr5_0] または [psu_cortexr5_1] を選択します。
 - Zynq-7000 SoC デバイス (zynq) には、Cortex-A9 のみがサポートされています。
[ps7_cortexa9_1] を選択します。
 - [Board Support Package] で次のいずれかを選択します。
 - 既に BSP を使用してアプリケーションを作成しており、同じ BSP を再利用する場合は、[Use existing] をオンにします。
 - 新しい BSP を作成するには、[Create New] をオンにします。



重要: [Create New] をオンにした場合、openamp ライブラリは自動的に組み込まれますが、後の手順に従ってコンパイラフラグを設定する必要があります。

- d. [Next] をクリックして、利用可能なテンプレートを選択します ([Finish] はクリックしない)。
4. [Available Templates] から、OpenAMP リモート ベアメタル用に利用可能な次の 3 つのアプリケーション テンプレートのうち 1 つを選択します。
 - OpenAMP echo-test
 - OpenAMP matrix multiplication Demo
 - OpenAMP RPC Demo
5. [Finish] をクリックします。
6. ザイリンクス Vitis の [Project Explorer] ビューで BSP を右クリックし、[Board Support Package Settings] を選択します。
7. [Board Support Package Settings] で [Overview] をクリックして [openamp] をオンにします。
8. [OK] をクリックします。

OpenAMP ザイリンクス Vitis の主要なソース ファイル

ザイリンクス Vitis アプリケーションでは、次の主要なソース ファイルが利用できます。

- **プラットフォーム情報** (platform_info.c/.h): これらのファイルには、OpenAMP に必要な情報を取得するために使用される、ハードコードされたプラットフォーム固有の値が含まれています。
 - #define IPI_IRQ_VECT_ID: プロセッサ間通信に使用される IPI エージェントのプロセッサ間割り込み (IPI) ベクター。
 - #define IPI_BASE_ADDR: プロセッサ間通信に使用される IPI エージェントのベース アドレス。
 - #define IPI_CHN_BITMASK: リモート プロセッサの IPI ビットマスク。どのリモート プロセッサと通信するかはビットマスクによって識別されるため、この値が必要です。ビットマスクの情報は『Zynq UltraScale+ MPSoC レジスタ リファレンス』に記載されています。
https://japan.xilinx.com/html_docs/registers/ug1087/ug1087-zynq-ultrascale-registers.html#_overview.html
- **リソース テーブル** (rsc_table.c/.h): リソース テーブルには、メモリ リソースと virtIO デバイス リソースを指定するエントリが含まれます。virtIO デバイスには、デバイスの機能、vring アドレス、サイズ、およびアライメント情報が含まれます。リソース テーブルのエントリは rsc_table.c で指定され、remote_resource_table 構造体は rsc_table.h で指定されます。
- **ヘルパー** (helper.c/.h): ヘルパーには、リモート アプリケーションがハードウェアと通信できるようにするプラットフォーム固有の API が含まれます。これには GIC の初期化および制御機能が含まれます。
- **アプリケーション コード** (src/<application>.c): XVitis 内のアプリケーションの src ディレクトリには、特定のアプリケーションが置かれます (rpmsg-echo.c/matrix_multiply.c/rpc_demo.c)。

カーネル空間での RPMsg を使用する Linux アプリケーションの構築

OpenAMP を使用した PetaLinux のセットアップ

PetaLinux を使用する前に、次の準備が必要です。

1. 適切なディレクトリに PetaLinux マスタープロジェクトを (パスにスペースを入れずに) 作成します。このガイドでは、プロジェクト名は `<plnx-proj-root>` です。

```
$ petalinux-create -t project -s <PATH_TO_PETALINUX_ZYNQMP_PROJECT_BSP>
```

2. `<plnx-proj-root>` ディレクトリに移動します。

```
$ cd <plnx-proj-root>
```

3. PetaLinux プロジェクトにリモート アプリケーションを組み込みます。

PetaLinux BSP に付属の、あらかじめ構築されたリモート ファームウェアを使用しない場合は、この手順が必要です。(たとえば、ザイリンクス Vitis による) リモート アプリケーションの開発と構築の完了後、そのアプリケーションが Linux ファイル システムから `remoteproc` を利用できるようにするには、アプリケーションは PetaLinux プロジェクトに組み込まれている必要があります。

- a. 次のコマンドを使用して、`components/apps/<app_name>` ディレクトリ内で PetaLinux アプリケーションを作成します。

```
$ petalinux-create -t apps --template install -n <app_name> --enable
```

- b. ザイリンクス Vitis で構築されたリモート プロセッサ用ファームウェア (すなわち、`.elf` ファイル) を、このディレクトリにコピーします。

```
project-spec/meta-user/recipes-apps/<app_name>/files/
```

- c. `project-spec/meta-user/recipes-apps/<app_name>/<app_name>.bb` を変更して、リモート プロセッサのファームウェアを RootFS にインストールします。

次に例を挙げます。

```
SUMMARY = "Simple test application"
SECTION = "PETALINUX/apps"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://<myfirmware>"
S = "${WORKDIR}"
INSANE_SKIP_${PN} = "arch"

do_install() {
    install -d ${D}/lib/firmware
    install -m 0644 ${S}/<myfirmware> ${D}/lib/firmware/<myfirmware>
}

FILES_${PN} = "/lib/firmware/<myfirmware>"
```

4. すべてのデバイスについて、OpenAMP と連携して動作するようにカーネル オプションを設定します。

- a. PetaLinux カーネル コンフィギュレーション ツールを起動します。

```
petalinux-config -c kernel
```

- b. ロード可能なモジュールのサポートを有効にします。
[*] Enable loadable module support --->
- c. remoteproc ドライバーのサポートを有効にします。コマンドは、使用する Zynq デバイスによって異なります。

```
Device Drivers --->
  Remoteproc drivers --->
    # for R5:
    <M> ZynqMP_r5 remoteproc support
    # for Zynq A9
    <M> Support ZYNQ remoteproc
```

5. RootFS 内のモジュールとアプリケーションをすべて有効にします。



重要: これらのオプションは、PetaLinux リファレンス BSP 内でのみ利用可能です。この手順のアプリケーションは、ユーザーが使用できる例です。

- a. RootFS コンフィギュレーション メニューを開きます。
petalinux-config -c rootfs
- b. OpenAMP アプリケーションと rpmsg モジュールが有効になっていることを確認します。

```
Filesystem Packages --->
-> Petalinux Package Groups
-> packagegroup-petalinux-openamp
```

注記: packagegroup-petalinux-openamp は、openamp に関連する多くのサブコンポーネントを有効にします。よりきめ細かな制御が必要な場合は、このパッケージグループを設定せずに、必要に応じて次のコンポーネントを個別に有効にしてください。

```
rpmsg-echo-test, rpmsg-mat-mul, rpmsg-proxy-app
```

上記のコンポーネントの各パッケージのソースコードへのリンクを次に示します。

- rpmsg-echo-test:
<https://github.com/Xilinx/meta-openamp/tree/rel-v2019.2/recipes-openamp/rpmsg-examples/rpmsg-echo-test>
- rpmsg-proxy-app:
https://github.com/Xilinx/meta-openamp/blob/rel-v2019.2/recipes-openamp/rpmsg-examples/rpmsg-proxy-app/proxy_app.c
- 必要に応じて、リモート プロセッサのデフォルト ファームウェア イメージの組み込みを有効にします。

```
Filesystem Packages --->
  misc --->
    openamp-fw-echo-testd --->
      [*] openamp-fw-echo-testd
    openamp-fw-mat-muld --->
      [*] openamp-fw-mat-muld
    openamp-fw-rpc-demo --->
      [*] openamp-fw-rpc-demo
```

注記: これにより、rootfs /lib/firmware ディレクトリに置かれるあらかじめ作成されたイメージで提供される、同じリモート プロセッサ ファームウェアが組み込まれます。ザイリンクス Vitis を使用して新しいイメージを作成する場合は、この手順は不要です。

デバイス ツリー バイナリ ソースの設定

PetaLinux リファレンス BSP には、次の場所に置かれた OpenAMP 用 DTB (デバイス ツリー バイナリ) が含まれています。

```
pre-built/linux/images/openamp.dtb
```

共有メモリおよびカーネル remoteproc 用のデバイス ツリーの設定例は、次の場所にあります。

```
project-spec/meta-user/recipes-bsp/device-tree/files/openamp.dtsi
```

openamp.dtb および openamp.dtsi ファイルは、参照用にのみ提供されています。system-user.dtsi ファイルを編集して、ユーザーのプロジェクトに openamp.dtsi からのコンテンツを組み込む必要があります。

オーバーレイには、OpenAMP がデバイス ツリー内で必要とするノードが含まれています。

- このデバイス ツリーの例は ZynqMP の場合です。

```
/ {
    reserved-memory {
        #address-cells = <2>;

        #size-cells = <2>;

        ranges;

        rpu0vdev0vring0: rpu0vdev0vring0@3ed40000 {

            no-map;

            reg = <0x0 0x3ed40000 0x0 0x4000>;

        };

        rpu0vdev0vring1: rpu0vdev0vring1@3ed44000 {

            no-map;

            reg = <0x0 0x3ed44000 0x0 0x4000>;

        };

        rpu0vdev0buffer: rpu0vdev0buffer@3ed48000 {

            no-map;

            reg = <0x0 0x3ed48000 0x0 0x100000>;

        };

        rproc_0_reserved: rproc@3ed00000 {

            no-map;

            reg = <0x0 0x3ed00000 0x0 0x40000>;

        };
    };
}
```



```
};

zynqmp-rpu {
    compatible = "xlnx,zynqmp-r5-remoteproc-1.0";

    #address-cells = <2>;

    #size-cells = <2>;

    ranges;

    core_conf = "split";

    reg = <0x0 0xFF9A0000 0x0 0x10000>;

    r5_0: r5@0 {
        #address-cells = <2>;

        #size-cells = <2>;

        ranges;

        memory-region = <&rproc_0_reserved>, <&rpu0vdev0buffer>, <&rpu0vdev0vring0>,
        <&rpu0vdev0vring1>;

        pnode-id = <0x7>;

        mboxs = <&ipi_mailbox_rpu0 0>, <&ipi_mailbox_rpu0 1>;

        mbox-names = "tx", "rx";

        tcm_0_a: tcm_0@0 {
            reg = <0x0 0xFFE00000 0x0 0x10000>;

            pnode-id = <0xf>;
        };

        tcm_0_b: tcm_0@1 {
            reg = <0x0 0xFFE20000 0x0 0x10000>;

            pnode-id = <0x10>;
        };
    };
};

};
```

```
zynqmp_ipi1 {
    compatible = "xlnx,zynqmp-ipi-mailbox";

    interrupt-parent = <&gic>;

    interrupts = <0 29 4>;

    xlnx,ipi-id = <7>;

    #address-cells = <1>;

    #size-cells = <1>;

    ranges;

    /* APU<->RPU0 IPI mailbox controller */
    ipi_mailbox_rpu0: mailbox@ff990600 {

        reg = <0xff990600 0x20>,
            <0xff990620 0x20>,
            <0xff9900c0 0x20>,
            <0xff9900e0 0x20>;

        reg-names = "local_request_region",
            "local_response_region",
            "remote_request_region",
            "remote_response_region";

        #mbox-cells = <1>;

        xlnx,ipi-id = <1>;

    };

};

};
```

デバイス ツリー用のメールボックスの使用方法については、デバイス ツリーの資料 (<https://github.com/Xilinx/linux-xlnx/blob/master/Documentation/devicetree/bindings/mailbox/xlnx%2czynqmp-ipi-mailbox.txt>) を参照してください。

ZynqMP での remoteproc ドライバーの使用方法については、デバイス ツリーの資料 (<https://github.com/Xilinx/linux-xlnx/blob/master/Documentation/devicetree/bindings/remoteproc/xilinx%2czynqmp-r5-remoteproc.txt>) を参照してください。

注記: Linux 上で動作する OpenAMP は、デフォルト IPI の使用をサポートしていません。Linux 上で動作する OpenAMP 用の IPI コンフィギュレーションは、デバイス ツリー内で設定されます。IPI については、『Zynq UltraScale+ MPSoC レジスタ リファレンス』(UG1087) の IPI モジュールを参照してください (https://japan.xilinx.com/html_docs/registers/ug1087/ug1087-zynq-ultrascale-registers.html)。

デフォルトの APU IPI は PMU FW との通信専用になっているため、上記のデバイス ツリー デモでは、APU 内の OpenAMP はプロセッサ間通知に (デフォルトの APU IPI の代わりに) PL0 IPI を使用します。

ZynqMP では、Cortex-R5 がどのように動作するかを `core_conf` パラメーターで設定できます。現在の設定は、このガイドで参照しているデモ アプリケーションで有効です。付録 A 「libmetal API」では、これらのパラメーターについてより詳しく説明しています。

- Zynq_A9 の場合、次のようになります。

```
/ {
    reserved-memory {
        #address-cells = <1>;
        #size-cells = <1>;

        ranges;

        vdev0vring0: vdev0vring0@3e800000 {
            no-map;

            compatible = "shared-dma-pool";

            reg = <0x3e800000 0x4000>;
        };

        vdev0vring1: vdev0vring1@3e804000 {
            no-map;

            compatible = "shared-dma-pool";

            reg = <0x3e804000 0x4000>;
        };

        vdev0buffer: vdev0buffer@3e808000 {
            no-map;

            compatible = "shared-dma-pool";

            reg = <0x3e808000 0x100000>;
        };

        rproc_0_reserved: rproc@3e000000 {
            no-map;
```

```

        compatible = "shared-dma-pool";

        reg = <0x3e000000 0x800000>;

    };

};

remoteproc0: remoteproc@0 {

    compatible = "xlnx,zynq_remoteproc";

    firmware = "firmware";

    vring0 = <15>;

    vring1 = <14>;

    memory-region = <&rproc_0_reserved>, <&vdev0buffer>, <&vdev0vring0>,
<&vdev0vring1>;

};

};

```

アプリケーションと Linux プロジェクトの構築

アプリケーションと Linux プロジェクトを構築するには、次の手順を実行します。

1. PetaLinux プロジェクトの root ディレクトリにいることを確認します。
`cd <plnx_proj>`
2. PetaLinux を構築します。petalinux-build



ヒント: 問題が発生した場合は、petalinux-build に -v を追加して、それぞれの文字出力を確認します。

構築が成功した場合、イメージは次の images/linux フォルダに置かれます。
<plnx_proj>/images/linux

PetaLinux プロジェクトのブート

PetaLinux プロジェクトは、QEMU (クイック エミュレーター) またはハードウェアからブートできます。

QEMU 上でのブート

構築の成功後、次のように QEMU 上で PetaLinux プロジェクトを実行できます。

1. PetaLinux ディレクトリに移動します。cd <plnx_proj>
2. PetaLinux ブートを実行します。petalinux-boot --qemu --kernel

注記: QEMU 上での OpenAMP のブートは、ZynqMP の場合にのみ有効です。

ハードウェア上でのブート

ビルドが問題なく完了したら、次のようにハードウェア上で PetaLinux プロジェクトを実行できます。次の手順に従って、ボード上で OpenAMP をブートします。

ボードのセットアップ

1. コンピューターにボードを接続し、ボードに電源が投入されていることを確認します。
2. ボードがリモート システムに接続されている場合は、リモート システム上で `hw_server` を起動します。
3. コンソール ターミナルを開き、ボード上の UART (Universal Asynchronous Receiver/Transmitter) に接続します。

イメージのダウンロード

1. PetaLinux ディレクトリに移動します。

```
cd <plnx_proj>
```
2. PetaLinux ブートを実行します。
 - 。 リモート システムを使用する場合:

```
petalinux-boot --jtag --kernel --hw_server-url <remote_system>
```
 - 。 ローカル システムを使用する場合:

```
petalinux-boot --jtag --kernel -bitstream <bitstream>
```



ヒント: 問題が発生した場合は、上記のコマンドに `-v` を追加して、出力される文字を確認します。

サンプル アプリケーションの実行

システムの起動後、ユーザー名 `root` とパスワード `root` を使用してログインします。ログイン後、次のサンプル アプリケーションが利用可能になります。

- [エコー テストの実行](#)
- [行列乗算テストの実行](#)
- [プロキシ アプリケーションの実行](#)

注記: 次のことに注意してください。

- 。 Linux カーネルのブートが完了すると、`remoteproc` ドライバーはロードされています。ドライバーがロードされていない場合は、カーネル コンフィギュレーションでこのドライバーが有効にされていることを確認し、デバイス ツリーを確認してください。
- 。 `remoteproc` ドライバーをアンロードしていた場合は、次のようにロードできます。
 - Zynq UltraScale+ MPSoC デバイスの場合:

```
modprobe zynqmp_r5_remoteproc
```
 - Zynq-7000 SoC デバイスの場合:

```
modprobe zynq_remoteproc
```

エコー テストの実行

1. エコー テスト ファームウェアと RPMsg モジュールをロードします。

```
echo image_echo_test > /sys/class/remoteproc/remoteproc0/firmware
echo start > /sys/class/remoteproc/remoteproc0/state
```
2. テストを実行します。

```
echo_test
```

テストが開始されます。
3. 画面上の指示に従ってテストを完了します。
4. テストの完了後、アプリケーションをアンロードします。

```
echo stop > /sys/class/remoteproc/remoteproc0/state
```

OpenAMP アプリケーションのデバッグ

RPU ファームウェアのデバッグ

ザイリンクス システム デバッガー (XSDB) を使用して RPU 0 上で実行されるサンプル エコー テストのデバッグ例を次に示します。この例では、`platform_info.c` の 295 行目に関数 `platform_init` があり、コンパイルされてアドレス `0x3ed011c8` に置かれます。次の例では、ブレークポイントを設定してそこまで実行し、ブレークポイントで停止するまでの範囲のローカル変数の値を出力する方法を示します。

```
xsdb% bpadd -addr 0x3ed011c8
0
xsdb% Info: Breakpoint 0 status:
    target 7: {Address: 0x3ed011c8 Type: Hardware}
xsdb% dow ~/test.elf
Downloading Program -- ~/test.elf
section, .vectors: 0x00000000 - 0x0000051f
section, .text: 0x3ed00000 - 0x3ed0d73f
section, .init: 0x3ed0d740 - 0x3ed0d74b
section, .fini: 0x3ed0d74c - 0x3ed0d757
section, .rodata: 0x3ed0d758 - 0x3ed0ee8f
section, .data: 0x00000520 - 0x00001623
section, .resource_table: 0x00001700 - 0x000017ff
section, .eh_frame: 0x3ed0ee90 - 0x3ed0ee93
section, .ARM.exidx: 0x3ed0ee94 - 0x3ed0ee9b
section, .init_array: 0x3ed0ee9c - 0x3ed0eea3
section, .fini_array: 0x3ed0eea4 - 0x3ed0eea7
section, .bss: 0x3ed0eea8 - 0x3ed0f157
section, .heap: 0x00001800 - 0x000057ff
section, .stack: 0x00005800 - 0x00008fff
100% OMB 0.3MB/s 00:00
Setting PC to Program Start Address 0x00000000
Successfully downloaded ~/test.elf
xsdb% con
xsdb% Info: Cortex-R5 #0 (target 7) Stopped at 0x3ed011c8 (Breakpoint)
platform_init() at ../src/platform_info.c: 295
295: {
xsdb% locals
argc      : 0
argv      : 0
```

```
platform : 0
proc_id : 0
rsc_id : 1053874736
rproc : 1053824852
xsdb% con
Info: Cortex-R5 #0 (target 7) Running
xsdb%
```

OpenAMP アプリケーションのデバッグ

PetaLinux 内でデバッグ シンボルを使用する OpenAMP Linux アプリケーションを生成するには、次の手順を実行します。

1. open-amp デモと、デバッグ シンボルを使用する open-amp を有効にします。これらは petalinux-config -c rootfs によって有効にできます。

```
-> Filesystem Packages
--> libs
---> open-amp
[*]open-amp
[*]open-amp-dbg
[*]open-amp-demos
```

2. gdb パッケージを有効にします。gdb パッケージは、次のように有効にできます。

```
petalinux-config -c rootfs
-> Filesystem Packages
--> misc
---> gdb
```

3. petalinux-build を使用して構築します。

gdb を使用して Linux アプリケーションをデバッグする方法については、gdb の資料 (<https://www.gnu.org/software/gdb/documentation>) を参照してください。

行列乗算テストの実行

1. 行列乗算ファームウェアと RPMsg モジュールをロードします。

```
echo image_matrix_multiply > /sys/class/remoteproc/remoteproc0/firmware
echo start > /sys/class/remoteproc/remoteproc0/state
```

2. テストを実行します。

```
mat_mul_demo
```

テストが開始されます。

3. 画面上の指示に従ってテストを完了します。
4. テストの完了後、アプリケーションをアンロードします。

```
echo stop > /sys/class/remoteproc/remoteproc0/state
```

プロキシ アプリケーションの実行

1. プロキシアプリケーションは、ロードと実行を 1 つの手順で行うことができ、必要なモジュールを自動的にロードします。

```
proxy_app
```

2. [Enter name] に対して、任意の文字列を入力します。
3. [Enter age] に対して、任意の整数を入力します。
4. [Enter value for pi] に対して、任意の浮動小数点数を入力します。
5. テストを [re-run] するように求められます。
6. アプリケーションの終了後、モジュールは自動的にアンロードされます。

Linux ユーザー空間の OpenAMP RPMsg を使用する Linux アプリケーションの構築

PetaLinux ツールを使用した Linux ユーザー空間の RPMsg デモ アプリケーションの構築

PetaLinux ツールを使用する前に、次の手順に従って準備します。

1. 適切なディレクトリに PetaLinux マスタープロジェクトを (パスにスペースを入れずに) 作成します。このガイドでは、プロジェクト名は <plnx_proj> です。

```
$ petalinux-create -t project -s <PATH_TO_PETALINUX_ZYNQMP_PROJECT_BSP>
```

2. プロジェクトディレクトリに移動します。

```
$ cd <plnx_proj>
```

3. rootfs コンフィギュレーションユーティリティを起動します。

```
$ petalinux-config -c rootfs
```

4. このデモに必要な rootfs パッケージを有効にします。

```
Filesystem Packages --->
misc --->
packagegroup-petalinux-openamp --->
[*] packagegroup-petalinux-openamp
```

注記: packagegroup-petalinux-openamp は、openamp に関連する多くのサブコンポーネントを有効にします。ここで必要なコンポーネントのみを有効にする場合は、このパッケージグループを設定せずに、次のコンポーネントを個別に有効にしてください。

```
open-amp、open-amp-demos、libmetal
```

5. Linux ユーザー空間の RPMsg アプリケーション デモ用にデバイス ツリーを設定します。

libmetal Linux デモは、IPI および共有メモリ用にユーザー空間の I/O (UIO) デバイスを使用します。PetaLinux プロジェクト内の <plnx-proj-root>/project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi に次のコードをコピーし、必要に応じて変更します。

```
/ {
    reserved-memory {
        #address-cells = <2>;
        #size-cells = <2>;
```



```

ranges;
rproc_0_reserved: rproc@3ed000000 {
    no-map;
    reg = <0x0 0x3ed00000 0x0 0x1000000>;
};

};

amba {
    /* Shared memory (APU to RPU) */
    shm0: shm@0 {
        compatible = "shm";
        reg = <0x0 0x3ed20000 0x0 0x0100000>;
    };

    /* IPI device */
    ipi0: ipi@0 {
        compatible = "ipi_uio";
        reg = <0x0 0xff340000 0x0 0x1000>;
        interrupt-parent = <&gic>;
        interrupts = <0 29 4>;
    };
};

};

```

注記: デフォルトの APU IPI は PMU FW 通信専用になっているため、OpenAMP は通信通知用にほかの IPI (PL0 IPI) を選択しました。

Linux ユーザー空間の RPMsg アプリケーション デモのソース コードは、次の場所に掲載されています。

- 3 つのアプリケーションに共通のコード:
- platform_info.c および platform_info.h は、プラットフォーム固有のデータを定義し、API をインプリメントして OpenAMP 用にプラットフォーム固有の情報を設定します。
 - https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp_r5/platform_info.c
 - https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp_r5/platform_info.h
- rsc_table.c および rsc_table.h は、リモート コアのリソース テーブルに情報を入力し、Linux マスターが使用できるようにします。
 - https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp_r5/rsc_table.c
 - https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp_r5/rsc_table.h
- アプリケーション固有のコード:
 - <https://github.com/OpenAMP/open-amp/blob/master/apps/examples/echo/rpmsg-echo.c>
 - https://github.com/OpenAMP/open-amp/blob/master/apps/examples/matrix_multiply/matrix_multiply.c
 - https://github.com/OpenAMP/open-amp/blob/master/apps/examples/rpc_demo/rpc_demo.c

注記: Linux 側の OpenAMP ユーザー空間アプリケーションが RPU1 と通信するようにするには、次のように更新します。

- https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp/platform_info.c#L34 の RPU_CPU_ID を 1 に変更します。
- https://github.com/OpenAMP/open-amp/blob/master/apps/machine/zynqmp/platform_info.c#L42 の 0x100 を 0x200 に変更します。
- RPU1 ファームウェアが 0x3EFX_XXXX で共有メモリを使い切る場合は、デバイス ツリーを更新し、共有メモリへの参照を 0x3EDX_XXXX から 0x3EFX_XXXX に更新します。

6. 「petalinux-build」を使用して PetaLinux プロジェクトを構築します。

```
$ petalinux-build
```

カーネル イメージとデバイス ツリー バイナリは、<plnx_proj>/images/linux ディレクトリに置かれます。

RPU ファームウェアの構築

注記: リソース テーブルのセクションが追加され、このセクションが予約済みメモリ内に配置されたことを示します。

1. ユーザー空間で使用される RPMsg の OpenAMP インプリメンテーションは静的な vring エントリのみを許容するため、rsc_table.c 内で RING_TX を FW_RSC_U32_ADDR_ANY から 0x3ed40000 へ、RING_RX を FW_RSC_U32_ADDR_ANY から 0x3ed44000 へ変更します。

```
#define RING_TX 0x3ed40000
#define RING_RX 0x3ed44000
```

boot.bin ファイルの作成

boot.bin ファイルを作成するには、例に示すように次のコマンドを実行して bif ファイルを渡します (bootgen.bif は bif ファイルの名前)。

```
$ petalinux-package --boot --bif
bootgen.bif --force
```

ハードウェア上でのテスト

1. PetaLinux プロジェクトに移動します。

```
$ cd <plnx_proj>
```

2. PetaLinux プロジェクトを構築します。

```
$ petalinux-build
```

3. ザイリンクス Vitis を使用して構築された RPU ファームウェアを、SD ブートでブートします。BIF ファイルの例を次に示します。

```
the_ROM_image:
{
    [fsbl_config] a53_x64
    [bootloader] <plnx_proj>/images/linux/zynqmp_fsbl.elf
    [destination_device=pl] <plnx_proj>/images/linux/system.bit
    [destination_cpu=pmu] <plnx_proj>/images/linux/pmufw.elf
    [destination_cpu=r5-0] <RPU firmware>
    [destination_cpu=a53-0, exception_level=el-3,
trustzone] <plnx_proj>/images/linux/arm/bl31.elf
```

```
[destination_cpu=a53-0,
exception_level=el-2] <plnx_proj>/images/linux/u-boot.elf
}
```

4. APU Linux ターゲット コンソール上で、デモ アプリケーション `rpmsg-echo-ping-shared`、`matrix_multiply-shared`、および `rpc_demod-shared` を実行します。このプロセスにより、次のような出力が生成されます。

```
root@xilinx-zcu102-2019_1:~# rpmsg-echo-ping-shared
metal: info:      metal_uio_dev_open: No IRQ for device 3ed20000.shm.
Successfulinitializing rpmsg vdev
Try to create rpmsg endpoint.
Successfully created rpmsg endpoint.
ly open shm device.
Successfully added shared memory
Successfully probed IPI device
Successfully initialized Linux r5 remoteproc.
Successfully initialized remoteproc
Calling mmap resource table.
Successfully mmap resource table.
Successfully set resource table to remoteproc.
Creating virtio...
Successfully created virtio device.
initializing rpmsg vdev
echo test: sent : 488
received payload number 471 of size 488
*****
Test Results: Error count = 0
*****
Quitting application .. Echo test end
rpmsg_channel_deleted
WARNING rx_vq: freeing non-empty virtqueue
WARNING tx_vq: freeing non-empty virtqueue
root@Xilinx-ZCU102-2019_1:~#

# matrix_multiply-shared
...
CLIENT> Matrix multiply: sent : 296
CLIENT> Quitting application .. Matrix multiplication end
CLIENT> *****
CLIENT> Test Results: Error count = 0
CLIENT> *****
CLIENT> rpmsg_channel_deleted
WARNING rx_vq: freeing non-empty virtqueue
WARNING tx_vq: freeing non-empty virtqueue
root@Xilinx-ZCU102-2019_1:~#

# rpc_demod-shared
login[1900]: root login on 'ttyPS0'
root@Xilinx-ZCU102-2019_1:~# proxy_app-openamp
...
Master> Remote proc resource initialized.
Master> RPMSG channel has created.
Remote>FreeRTOS Remote Procedure Call (RPC) Demonstration
Remote>*****
Remote>Rpmsg based retargetting to proxy initialized..
Remote>FileIO demo ..
```

```
Remote>Creating a file on master and writing to it..  
... ..  
Remote>Repeat demo ? (enter yes or no)  
no  
Remote>RPC retargetting quitting ...  
Remote> Firmware's rpmsg-openamp-demo-channel going down!  
Master>  
RPC service exiting !!  
Master> sending shutdown signal.  
WARNING rx_vq: freeing non-empty virtqueue  
WARNING tx_vq: freeing non-empty virtqueue  
root@Xilinx-ZCU102-2019_1:~#
```

システム設計に関する留意事項

この章では、OpenAMP と libmetal のさまざまな制限に関する情報を提供します。

サポートされるコンフィギュレーション

RPMsg カーネル空間とは、VirtIO、RPMsg および Remoteproc をインプリメントしているカーネルドライバです。RPMsg ユーザー空間とは、VirtIO、RPMsg および Remoteproc の OpenAMP インプリメンテーションです。

表 4-1: 機能

	APU 上の Linux カーネル RPMsg/Remoteproc + RPU 上で使用される OpenAMP ライブラリ	Linux ユーザー空間で使用される OpenAMP ライブラリ + RPU 上で 使用される OpenAMP ライブラリ	APU と RPU の両方で 使用される libmetal ライブラリ
Linux による RPU の ブート (RPU は Linux APU ホストのコプロセッサ)	あり 「PetaLinux イメージのク イックトライ」参照	あり 「Linux ユーザー空間の OpenAMP RPMsg を使用する Linux アプリ ケーションの構築」参照	あり 「ザイリンクス libmetal AMP デモ」参照
ウォーム リスタートの サポート: APU 再起動 後の APU/RPU の自動 再接続	あり 次を参照 http://www.wiki.xilinx.com/ OpenAMP	なし	ユーザー定義
事前に定義された共有 メモリ範囲のサポート	あり 「簡単な OpenAMP アプリ ケーションの作成方法」 参照	あり 「Linux ユーザー空間の OpenAMP RPMsg を使用する Linux アプリ ケーションの構築」参照	あり 「共有メモリ」および 「libmetal と PetaLinux ツールを使用した Linux デモ アプリケー ションの有効化」
Linux による共有メモリ 範囲の動的な割り当て	あり 「簡単な OpenAMP アプリ ケーションの作成方法」 参照	なし	なし
複数の通信チャネルの サポート (例: 両方の RPU)	あり 「OpenAMP デモ」参照	あり 「OpenAMP デモ」参照	あり 「OpenAMP デモ」参照

表 4-1: 機能

	APU 上の Linux カーネル RPMsg/Remoteproc + RPU 上で使用される OpenAMP ライブラリ	Linux ユーザー空間で使用される OpenAMP ライブラリ + RPU 上で 使用される OpenAMP ライブラリ	APU と RPU の両方で 使用される libmetal ライブラリ
FSBL RPU ブートへの 対応	なし	あり	あり 次を参照 http://www.wiki.xilinx.com/OpenAMP
データ転送オーバー ヘッド	ユーザー アプリケーショ ンと Linux カーネル間の メモリ コピーと、Linux カーネル空間から共有メ モリへのメモリ コピー	ユーザー アプリケーションと共有 メモリ間のメモリ コピー	

その他の留意事項

OpenAMP は、プロセッサ間通信用の Remoteproc、VirtIO および RPMsg のソース コードを提供しています。既に通信ソリューションをお持ちであるか、より軽いソリューションを使用する場合は、libmetal ライブラリを基盤として独自のソリューションを開発できます。

既知の制限

OpenAMP の既知の制限を次に示します。

- QEMU を使用した Zynq® デバイスの OpenAMP デモはサポートしていません。
QEMU を使用した Zynq UltraScale+™ MPSoC デバイスの OpenAMP デモのみをサポートしています。
- 共有メモリを Linux ユーザー空間の通常のメモリとして使用することはできません。Linux ユーザー空間の libmetal は UIO を使用するため、共有メモリはデバイス メモリとして使用する必要があります。
- APU 用に定義されたデフォルトの IPI は、Linux がパワー マネージメント機能に使用します。OpenAMP は、PL が使用するよう指定された IPI の中の 1 つを使用します。
- RPMsg のバッファ サイズは 512 バイトに制限され、そのうち 496 バイトはペイロードに使用されます。

Linux RPMsg のバッファ サイズ

OpenAMP のメッセージサイズは、rpmmsg カーネル モジュールによって定義されるバッファ サイズによって制限されます。Linux 4.19 カーネルでは、現在のところ 512 バイトと定義され、そのうち 16 バイトはメッセージ ヘッダー用、496 バイトはペイロード用です。



重要: RPMsg バッファ サイズは再定義しないでください。

libmetal API

libmetal API の関数

この付録で説明する libmetal API は、libmetal ユーザー向けの API です。libmetal 開発者が libmetal ライブラリを変更して独自のプラットフォーム/OS に libmetal を対応させたい場合は、内部 libmetal API の libmetal doxygen を参照してください。

最上位インターフェイス

metal_init

説明

libmetal ライブラリを初期化します。

引数

params: 初期化パラメーター

戻り値

成功した場合は 0、失敗した場合は -errno を返します。

用途

```
int metal_init(const struct metal_init_params params);
```

metal_finish

説明

libmetal ライブラリをシャットダウンし、すべての予約リソースを解放します。

用途

```
void metal_finish(void);
```

割り込み処理インターフェイス

metal_irq_handler

説明

割り込みハンドラーのタイプを示します。

引数

- irq: irq 割り込み id
- arg: ハンドラーに渡す引数

戻り値

irq 処理されたステータスを返します。

用途

```
typedef int (*metal_irq_handler) (int irq, void *arg);
```

metal_irq_register

説明

- 割り込みを登録するか、または特定の割り込みの割り込み処理を登録します。
- 割り込みハンドラー パラメーター (irq_handler) が NULL の場合は、割り込みハンドラーの登録を解除します。
- 割り込みハンドラー、デバイス (dev)、およびドライバ ID (drv_id) が NULL の場合は、その割り込みに対応するすべてのハンドラーの登録を解除します。
- 割り込みハンドラーが NULL で、デバイスまたは driver ID が NULL でない場合は、同じデバイスまたはドライバ ID を使用して登録された割り込みハンドラーのみ登録を解除します。

引数

irq	割り込み id
irq_handler	割り込みハンドラー
dev	この irq が所属するメタル デバイス
drv_id	ドライバ id。ドライバ データ用に使用できます。

戻り値

成功した場合は 0、失敗した場合は 0 でない値を返します。

用途

```
int metal_irq_register(int irq, metal_irq_handler irq_handler, void  
*arg); metal_irq_save_disable
```

説明

割り込みを無効にします。

戻り値

割り込みステート

用途

```
unsigned int metal_irq_save_disable(void);
```

metal_irq_restore_enable

説明

割り込みを前の状態に戻します。

引数

前の割り込みステートにフラグを立てます。

用途

```
void metal_irq_restore_enable(unsigned int flags);
```

metal_irq_enable

説明

指定された割り込みを有効にします。

引数

- ベクター
- 割り込みベクター番号

用途

```
void metal_irq_enable(unsigned int vector);
```

metal_irq_disable

説明

指定された割り込みを無効にします。

引数

- ベクター
- 割り込みベクター番号

用途

```
void metal_irq_disable(unsigned int vector);
```

共有メモリ インターフェイス

metal_shmem_open

説明

libmetal 共有メモリ セグメントを開きます。

引数

name	開くセグメントの名前
size	セグメントのサイズ
flags	キャッシュ可能にするかどうかを示す shmem フラグ

戻り値

成功した場合は 0、失敗した場合は -errno を返します。

用途

```
extern int metal_shmem_open(const char *name, size_t size, unsigned int flags, struct
metal_generic_shmem **result);metal_shmem_register_generic
```

説明

- 汎用共有メモリ領域を静的に登録します。
- 共有メモリ領域は、アプリケーションの初期化時に静的に登録することも、動的に開くこともできます。
- このインターフェイスは、領域の静的な登録に使用されます。
- これ以降に metal_shmem_open() を呼び出すと、この事前登録された領域のリストが検索されます。

引数

shmem: 汎用 shmem 構造体

戻り値

成功した場合は 0、失敗した場合は -errno を返します。

用途

```
extern int metal_shmem_register_generic(struct metal_generic_shmem *shmem);
```

スピンロック インターフェイス

metal_spinlock_init

説明

libmetal スピンロックを初期化します。

引数

slock: 初期化するスピンロック

用途

```
static inline void metal_spinlock_init(struct metal_spinlock *slock)
```

metal_spinlock_acquire

説明

スピンロックを取得します。

引数

slock: 取得するスピンロック

用途

```
static inline void metal_spinlock_acquire(struct metal_spinlock *slock)
```

metal_spinlock_release

説明

以前に取得したスピンロックを解放します。

引数

slock: 解放するスピンロック

用途

```
static inline void metal_spinlock_release(struct metal_spinlock *slock)
```

スリープ インターフェイス

metal_sleep_usec

説明

呼び出しスレッド内の次の実行を `usec` マイクロ秒遅延させます。

引数

`usec`: マイクロ秒の間隔

戻り値

成功した場合は 0、失敗した場合は 0 でない値を返します。

用途

```
int metal_sleep_usec(unsigned int usec);
```

ミューテックス インターフェイス

metal_mutex_init

説明

libmetal ミューテックスを初期化します。

引数

`mutex`: 初期化するミューテックス

用途

```
static inline void metal_mutex_init(metal_mutex_t *mutex); metal_mutex_deinit
```

説明

libmetal ミューテックスの初期化を解除します。

引数

`mutex`: 初期化を解除するミューテックス

用途

```
static inline void metal_mutex_deinit(metal_mutex_t *mutex);
```

metal_mutex_deinit

説明

libmetal ミューテックスの初期化を解除します。

引数

mutex: チェックするミューテックス

用途

```
static inline void metal_mutex_deinit(metal_mutex_t *mutex);
```

metal_mutex_try_acquire

説明

ミューテックスの取得を試みます。

引数

mutex: 取得するミューテックス

戻り値

取得に失敗した場合は 0、成功した場合は 0 でない値。

用途

```
static inline int metal_mutex_try_acquire(metal_mutex_t *mutex);
```

metal_mutex_acquire

説明

ミューテックスを取得します。

引数

mutex: 取得するミューテックス

用途

```
static inline void metal_mutex_acquire(metal_mutex_t *mutex);
```

metal_mutex_release

説明

以前に取得したミューテックスを解放します。

引数

mutex: 取得するミューテックス

用途

```
static inline void metal_mutex_release(metal_mutex_t *mutex);
```

metal_mutex_is_acquired

説明

ミューテックスが取得されたかどうかチェックします。

引数

mutex: チェックするミューテックス

用途

```
static inline int metal_mutex_is_acquired(metal_mutex_t *mutex);
```

I/O インターフェイス

metal_io_init

説明

libmetal の I/O 領域を開きます。

引数

io	I/O 領域ハンドル
virt	領域の仮想アドレス
physmap	ページごとの物理アドレスの配列
size	領域のサイズ
page_shift	ページサイズの Log2 (単一ページの場合は -1)。
mem_flags	メモリ フラグ
ops	ops

用途

```
static inline void metal_io_init(struct metal_io_region *io, void *virt, const
metal_phys_addr_t *physmap, size_t size, unsigned page_shift, unsigned int
mem_flags, const struct metal_io_ops *ops)
```

metal_io_finish

説明

libmetal 共有メモリ セグメントを閉じます。

引数

io: I/O 領域ハンドル

用途

```
static inline void metal_io_finish(struct metal_io_region *io)
```

metal_io_region_size

説明

I/O 領域のサイズを取得します。

引数

io: I/O 領域ハンドル

戻り値

I/O 領域のサイズ

用途

```
static inline size_t metal_io_region_size(struct metal_io_region *io)
```

metal_io_virt

説明

I/O 領域内の指定されたオフセットの仮想アドレスを取得します。

引数

- io: I/O 領域ハンドル
- offset: 共有メモリ セグメント内のオフセット

戻り値

オフセットが範囲外の場合は NULL、そうでない場合はオフセットへのポインター

用途

```
static inline void metal_io_virt(struct metal_io_region *io, unsigned long offset)
```

metal_io_virt_to_offset

説明

仮想アドレスを I/O 領域内のオフセットに変換します。

引数

- io: I/O 領域ハンドル
- virt: セグメント内の仮想アドレス

戻り値

範囲外の場合は METAL_BAD_OFFSET、そうでない場合はオフセット

用途

```
static inline unsigned long metal_io_virt_to_offset(struct metal_io_region *io, void
*virt)
```

metal_io_phys

説明

I/O 領域内の指定されたオフセットの物理アドレスを取得します。

引数

- io: I/O 領域ハンドル
- offset: 共有メモリ セグメント内のオフセット

戻り値

オフセットが範囲外の場合は METAL_BAD_PHYS、そうでない場合はオフセットの物理アドレス

用途

```
static inline metal_phys_addr_t metal_io_phys(struct metal_io_region *io, unsigned
long offset)
```

metal_io_phys_to_offset

説明

物理アドレスを I/O 領域内のオフセットに変換します。

引数

- io: I/O 領域ハンドル
- phys: セグメント内の物理アドレス

戻り値

範囲外の場合は METAL_BAD_OFFSET、そうでない場合はオフセット

用途

```
static inline unsigned long metal_io_phys_to_offset(struct metal_io_region *io,
metal_phys_addr_t phys)
```

metal_io_phys_to_virt

説明

物理アドレスを仮想アドレスに変換します。

引数

- io: 共有メモリ セグメント ハンドル
- phys: セグメント内の物理アドレス

戻り値

範囲外の場合は NULL、そうでない場合は対応する仮想アドレス

用途

```
static inline void metal_io_phys_to_virt(struct metal_io_region *io,
metal_phys_addr_t phys)
```

metal_io_virt_to_phys

説明

仮想アドレスを物理アドレスに変換します。

引数

- io: 共有メモリ セグメント ハンドル
- virt: セグメント内の仮想アドレス

戻り値

範囲外の場合は METAL_BAD_PHYS、そうでない場合は対応する物理アドレス

用途

```
static inline metal_phys_addr_t metal_io_virt_to_phys(struct metal_io_region *io,
void *virt)
```

metal_io_read

説明

I/O 領域から値を読み出します。

引数

- io: I/O 領域ハンドル
- offset: I/O 領域内のオフセット
- order: メモリの順序付け
- width: 読み出すデータタイプの幅 (バイト単位)。この関数がインラインとして適切に機能するには、この値は 1、2、4、または 8、およびコンパイル時定数でなければなりません。

戻り値

値

用途

```
static inline uint64_t metal_io_read(struct metal_io_region *io, unsigned long
offset, memory_order order, int width)
```

metal_io_write

説明

I/O 領域に値を書き込みます。

引数

- io: I/O 領域ハンドル
- offset: I/O 領域内のオフセット
- value: 書き込む値
- order: メモリの順序付け
- width: 書き込むデータタイプの幅 (バイト単位)。この関数がインラインとして適切に機能するには、この値は 1、2、4、または 8、およびコンパイル時定数でなければなりません。

用途

```
static inline void metal_io_write(struct metal_io_region *io, unsigned long offset,
uint64_t value, memory_order order, int width)
```

metal_io_block_read

説明

I/O 領域からブロックを読み出します。

引数

- io: I/O 領域ハンドル
- offset: I/O 領域内のオフセット
- dst: 読み出されたデータを格納するデスティネーション
- len: 読み出す長さ (バイト単位)

戻り値

成功した場合、読み出されたバイト数。失敗した場合、負の値。

用途

```
int metal_io_block_read(struct metal_io_region *io, unsigned long offset, void
*restrict dst, int len);
```

metal_io_block_write

説明

I/O 領域にブロックを書き込みます。

引数

- io: I/O 領域ハンドル
- offset: I/O 領域内のオフセット
- src: 書き込むソース
- len: 書き込む長さ (バイト単位)

戻り値

成功した場合、書き込まれたバイト数。失敗した場合、負の値。

用途

```
int metal_io_block_write(struct metal_io_region *io, unsigned long offset, const
void *restrict src, int len);
```

metal_io_block_set

説明

I/O 領域のブロックを充填します。

引数

- io: I/O 領域ハンドル
- offset: I/O 領域内のオフセット
- value: ブロックに充填する値
- len: 充填する長さ (バイト単位)

戻り値

成功した場合、充填されたバイト数。失敗した場合、負の値。

用途

```
int metal_io_block_set(struct metal_io_region *io, unsigned long offset, unsigned
char value, int len);
```

バス アブストラクション

metal_bus_register

説明

libmetal バスを登録します。

引数

bus: 事前に初期化されたバス構造

戻り値

成功した場合は 0、失敗した場合は -errno

用途

```
extern int metal_bus_register(struct metal_bus *bus);
```

metal_bus_unregister

説明

libmetal バスの登録を解除します。

引数

bus: 事前に登録されたバス構造

戻り値

成功した場合は 0、失敗した場合は -errno

用途

```
extern int metal_bus_unregister(struct metal_bus *bus);
```

metal_bus_find

説明

libmetal バスを名前で検索します。

引数

- name: バス名
- bus: 返されるバス ハンドル

戻り値

成功した場合は 0、失敗した場合は -errno

用途

```
extern int metal_bus_find(const char *name, struct metal_bus **bus);
```

metal_register_generic_device

説明

汎用 libmetal デバイスを静的に登録します。デバイスは、アプリケーションの初期化時に静的に登録することも、sysfs または libfdt ベースのエNUMレーションによって実行時に動的に開くこともできます。このインターフェイスは、デバイスの静的な登録に使用されます。これ以降に metal_device_open() を呼び出すと、この「汎用バス」上の事前登録されたデバイスのリストが検索されます。

引数

device: 汎用デバイス

戻り値

成功した場合は 0、失敗した場合は -errno

用途

```
extern int metal_register_generic_device(struct metal_device *device);
```

metal_device_open

説明

libmetal デバイスを名前で開きます。

引数

- bus_name: バス名
- devi_name: デバイス名
- device: 返されるデバイス ハンドル

戻り値

成功した場合は 0、失敗した場合は -errno

用途

```
extern int metal_device_open(const char *bus_name, const char *dev_name, struct metal_device **device);
```

metal_device_close

説明

libmetal デバイスを閉じます。

引数

device: デバイス ハンドル

用途

```
extern void metal_device_close(struct metal_device *device);
```

metal_device_io_region

説明

デバイス領域の I/O 領域アクセサーを取得します。

引数

- device: デバイス ハンドル
- index: 領域インデックス

戻り値

I/O アクセサー ハンドル、失敗した場合は NULL

用途

```
static inline struct metal_io_region metal_device_io_region(struct metal_device  
*device, unsigned index)
```

条件変数インターフェイス

metal_condition_init

説明

libmetal 条件変数を初期化します。

引数

cv: 初期化する条件変数

用途

```
static inline void metal_condition_init(struct metal_condition *cv);
```

metal_condition_signal

説明

1 つの waiter に通知します。この関数を呼び出す前に、呼び出し元がミューテックスを取得している必要があります。

引数

cv: 条件変数

戻り値

エラーがない場合は 0、エラーが発生した場合は 0 でない値

用途

```
static inline int metal_condition_signal(struct metal_condition *cv);
```

metal_condition_broadcast

説明

すべての waiter に通知します。この関数を呼び出す前に、呼び出し元がミューテックスを取得している必要があります。

引数

cv: 条件変数

戻り値

エラーがない場合は 0、エラーが発生した場合は 0 でない値

用途

```
static inline int metal_condition_broadcast(struct metal_condition *cv);
```

metal_condition_wait

説明

条件変数が通知されるまでブロックします。この関数を呼び出す前に、呼び出し元がミューテックスを取得している必要があります。

引数

- cv: 条件変数
- m: ミューテックス

戻り値

成功した場合は 0、失敗した場合は 0 でない値

用途

```
int metal_condition_wait(struct metal_condition *cv, metal_mutex_t *m);
```

アロケーション インターフェイス

metal_allocate_memory

説明

要求されたメモリ サイズを割り当てます。割り当てられたメモリへのポインターを返します。

引数

size: 要求されたメモリのサイズ (バイト単位)

戻り値

メモリ ポインター、割り当てに失敗した場合は 0

用途

```
static inline void *metal_allocate_memory(unsigned int size);
```

metal_free_memory

説明

以前に割り当てられたメモリを解放します。

引数

ptr: メモリへのポインター

用途

```
static inline void metal_free_memory(void *ptr);
```

ライブラリ バージョン インターフェイス

metal_ver_major

説明

ライブラリのメジャー バージョン番号を示します。メジャー バージョン番号を返します。METAL_VER_MAJOR (アプリケーションがコンパイルされたライブラリのメジャー バージョン) の値と一致する必要があります。

戻り値

アプリケーションにリンクされたライブラリのメジャー バージョン番号

用途

```
extern int metal_ver_major(void);
```


metal_ver_minor

説明

ライブラリのマイナーバージョン番号を示します。METAL_VER_MINOR (アプリケーションがコンパイルされたライブラリのマイナーバージョン) の値とは異なることがあります。

戻り値

アプリケーションにリンクされたライブラリのマイナーバージョン番号

用途

```
extern int metal_ver_minor(void);
```

metal_ver_patch

説明

ライブラリのパッチレベルを示します。METAL_VER_PATCH (アプリケーションがコンパイルされたライブラリのパッチレベル) の値とは異なることがあります。

戻り値

アプリケーションにリンクされたライブラリのパッチレベル

用途

```
extern int metal_ver_patch(void);
```

metal_ver

説明

ライブラリのバージョン文字列を示します。METAL_VER (アプリケーションがコンパイルされたライブラリのバージョン文字列) の値とは異なることがあります。

戻り値

アプリケーションにリンクされたライブラリのバージョン文字列

用途

```
extern const char *metal_ver(void);
```

OpenAMP API

Remoteproc API

はじめに

OpenAMP フレームワークで提供される remoteproc API により、マスター上のソフトウェア アプリケーションは、リモート プロセッサおよびそれに関連するソフトウェアを管理できます。

この章では、OpenAMP ライブラリ内の remoteproc インプリメンテーションについて紹介し、remoteproc API およびワークフローの概要を示します。

remoteproc API の関数

remoteproc_init

説明

remoteproc インスタンスを初期化します。

用途

```
struct remoteproc *remoteproc_init(struct remoteproc *rproc,  
                                   struct remoteproc_ops *ops, void *priv);
```

引数

rproc	remoteproc インスタンスへのポインター
ops	remoteproc 演算へのポインター
priv	プライベート データへのポインター

戻り値

作成された remoteproc ポインター。

remoteproc_remove

説明

remoteproc インスタンスを削除します。

用途

```
int remoteproc_resource_remove(struct remoteproc *rproc);
```

引数

rproc - remoteproc インスタンスへのポインター

戻り値

戻り値なし。

remoteproc_get_io_with_name

説明

この関数は、名前を指定して remoteproc メモリ I/O 領域を取得します。

用途

```
struct metal_io_region *  
remoteproc_get_io_with_name(struct remoteproc *rproc,  
                             const char *name);
```

引数

rproc - リモート プロセッサへのポインター

name - 共有メモリの名前

戻り値

libmetal I/O 領域ポインター、失敗した場合は NULL。

remoteproc_get_io_with_pa

説明

この関数は、物理アドレスを指定して remoteproc メモリ I/O 領域を取得します。

用途

```
struct metal_io_region *  
remoteproc_get_io_with_pa(struct remoteproc *rproc,  
                           metal_phys_addr_t pa);
```

引数

rproc - リモート プロセッサへのポインター

pa - 物理アドレス

戻り値

libmetal I/O 領域ポインター、失敗した場合は NULL。

remoteproc_get_io_with_da

説明

この関数は、デバイス アドレスを指定して remoteproc メモリ I/O 領域を取得します。

用途

```
struct metal_io_region *  
remoteproc_get_io_with_da(struct remoteproc *rproc,  
                           metal_phys_addr_t da,  
                           unsigned long *offset);
```

引数

rproc	リモート プロセッサへのポインター
da	物理アドレス
offset	デバイス アドレスの I/O 領域オフセット

戻り値

libmetal I/O 領域ポインター、失敗した場合は NULL。

remoteproc_get_io_with_va

説明

この関数は、仮想アドレスを指定して remoteproc メモリ I/O 領域を取得します。

用途

```
struct metal_io_region *
remoteproc_get_io_with_va(struct remoteproc *rproc,
                          void *va);
```

引数

rproc - リモート プロセッサへのポインター

va - 仮想アドレス

戻り値

libmetal I/O 領域ポインター、失敗した場合は NULL。

remoteproc_mmap

説明

この関数は、メモリを mmap するように remoteproc に指示します。

用途

```
void *remoteproc_mmap(struct remoteproc *rproc,
                      metal_phys_addr_t *pa, metal_phys_addr_t *da,
                      size_t size, unsigned int attribute,
                      struct metal_io_region **io);
```

引数

rproc	リモート プロセッサへのポインター
pa	物理アドレス ポインター
da	デバイス アドレス ポインター
size	メモリのサイズ
attribute	メモリの属性
io	I/O 領域へのポインター

戻り値

メモリへのポインターを返します。

remoteproc_parse_rsc_table

説明

この関数は、remoteproc のリソース テーブルを解析します。

用途

```
int remoteproc_parse_rsc_table(struct remoteproc *rproc,  
                               struct resource_table *rsc_table,  
                               size_t rsc_size);
```

引数

rproc	リモート インスタンスへのポインター
rsc_table	リソース テーブルへのポインター
rsc_size	リソース テーブルのサイズ

戻り値

成功した場合は 0、エラーが発生した場合は負の値を返します。

remoteproc_set_rsc_table

説明

この関数は、remoteproc のリソース テーブルの解析と設定を実行します。

用途

```
int remoteproc_set_rsc_table(struct remoteproc *rproc,  
                              struct resource_table *rsc_table,  
                              size_t rsc_size);
```

引数

rproc	リモート インスタンスへのポインター
rsc_table	リソース テーブルへのポインター
rsc_size	リソース テーブルのサイズ

戻り値

成功した場合は 0、エラーが発生した場合は負の値を返します。

remoteproc_create_virtio

説明

この関数は、virtio デバイスを作成し、作成された virtio デバイスへのポインターを返します。

用途

```
struct virtio_device *  
remoteproc_create_virtio(struct remoteproc *rproc,  
    int vdev_id, unsigned int role,  
    void (*rst_cb)(struct virtio_device *vdev));
```

引数

rproc	remoteproc インスタンスへのポインター
vdev_id	virtio デバイス ID
role	virtio デバイスの役割
rst_cb	virtio デバイスのリセット コールバック

戻り値

作成された virtio デバイスへのポインターを返し、失敗した場合は NULL を返します。

remoteproc_remove_virtio

説明

この関数は、virtio デバイスを削除します。

用途

```
void remoteproc_remove_virtio(struct remoteproc *rproc,  
    struct virtio_device *vdev);
```

引数

rproc	リモート インスタンスへのポインター
vdev	virtio デバイスへのポインター

戻り値

戻り値なし。

remoteproc_get_notification

説明

この関数は、remoteproc を通知し、remoteproc のサブデバイスを通知するかどうかチェックします。

用途

```
int remoteproc_get_notification(struct remoteproc *rproc,  
                               uint32_t notifyid);
```

引数

rproc - リモート インスタンスへのポインター

notifyid - 通知 ID

戻り値

成功した場合は 0、エラーが発生した場合は負の値を返します。

RPMmsg の開発

概要

OpenAMP フレームワークで提供される RPMmsg API により、マスターまたはリモート プロセッサ上で実行されるベアメタルまたは RTOS アプリケーションは、AMP 構成でプロセス間割り込み (IPI) を実行できます。この情報は、rpmmsg.h および rpmmsg_virtio.h ヘッダー ファイルで提供される資料に基づいています。

この章では、OpenAMP ライブラリ内の RPMmsg インプリメンテーションについて紹介し、RPMmsg API およびワークフローの概要を示します。

RPMmsg API の関数

rpmmsg_send_offchannel_raw()

説明

ソース アドレスとデスティネーション アドレスを指定して、リモート プロセッサにメッセージを送信します。この関数は、ソース src アドレスからリモート dst アドレスに長さ len のデータを送信します。メッセージは、チャンネルが所属するリモート プロセッサに送信されます。

利用可能な TX バッファがない場合、いずれかのバッファが利用可能になるか、または 15 秒経過してタイムアウトになるまで、この関数はブロックされます。タイムアウトになった場合は、-ERESTARTSYS が返されます。

用途

```
int rpmsg_send_offchannel_raw(struct rpmsg_endpoint *ept, uint32_t src,
                             uint32_t dst, const void *data, int size,
                             int wait)
```

引数

ept	RPMsg エンドポイント
data	メッセージのペイロード
len	ペイロードの長さ

戻り値

送信したバイト数、失敗した場合は負のエラー値を返します。

rpmsg_send()

説明

リモート プロセッサにメッセージを送信します。この関数は、ept に基づいて長さ len のデータを送信します。メッセージは、ept のソース アドレスとデスティネーション アドレスを使用して、チャンネルが所属するリモート プロセッサに送信されます。

利用可能な TX バッファがない場合、いずれかのバッファが利用可能になるか、または 15 秒経過してタイムアウトになるまで、この関数はブロックされます。タイムアウトになった場合は、-ERESTARTSYS が返されます。

用途

```
static inline int rpmsg_send(struct rpmsg_endpoint *ept, const void *data,
                             int len)
```

引数

ept	RPMsg エンドポイント
data	メッセージのペイロード
len	ペイロードの長さ

戻り値

送信したバイト数、失敗した場合は負のエラー値を返します。

rpmsg_sendto()

説明

リモート プロセッサにメッセージを送信します。この関数は、`ept` に基づいて長さ `len` のデータを送信します。メッセージは、`ept` のソース アドレスとデスティネーション アドレスを使用して、チャンネルが所属するリモート プロセッサに送信されます。

利用可能な TX バッファがない場合、いずれかのバッファが利用可能になるか、または 15 秒経過してタイムアウトになるまで、この関数はブロックされます。タイムアウトになった場合は、`-ERESTARTSYS` が返されます。

用途

```
static inline int rpmsg_send(struct rpmsg_endpoint *ept, const void *data,
                             int len)
```

引数

<code>ept</code>	RPMsg エンドポイント
<code>data</code>	メッセージのペイロード
<code>len</code>	ペイロードの長さ

戻り値

送信したバイト数、失敗した場合は負のエラー値を返します。

rpmsg_send_offchannel()

説明

明示的な `src/dst` アドレスを使用してメッセージを送信します。この関数は、長さ `len` のデータをリモート `dst` アドレスに送信し、ソース アドレスとして `src` を使用します。メッセージは、`ept` チャンネルが所属するリモート プロセッサに送信されます。

利用可能な TX バッファがない場合、いずれかのバッファが利用可能になるか、または 15 秒経過してタイムアウトになるまで、この関数はブロックされます。タイムアウトになった場合は、`-ERESTARTSYS` が返されます。

用途

```
static inline int rpmsg_send_offchannel(struct rpmsg_endpoint *ept,
                                         uint32_t src, uint32_t dst,
                                         const void *data, int len)
```

引数

<code>ept</code>	RPMsg エンドポイント
<code>src</code>	ソース アドレス
<code>dst</code>	デスティネーション アドレス
<code>data</code>	メッセージのペイロード
<code>len</code>	ペイロードの長さ

戻り値

送信したバイト数、失敗した場合は負のエラー値を返します。

`rpmsg_trysend()`

説明

リモート プロセッサにメッセージを送信します。この関数は、`ept` チャンネルに基づいて長さ `len` のデータを送信します。メッセージは、`ept` のソース アドレスとデスティネーション アドレスを使用して、`ept` チャンネルが所属するリモート プロセッサに送信されます。

利用可能な TX バッファがない場合、この関数はいずれかのバッファが利用可能になるまで待たずに、直ちに `-ENOMEM` を返します。

用途

```
static inline int rpmsg_trysend(struct rpmsg_endpoint *ept, const void *data,  
                                int len)
```

引数

<code>ept</code>	RPMsg エンドポイント
<code>data</code>	メッセージのペイロード
<code>len</code>	ペイロードの長さ

戻り値

送信したバイト数、失敗した場合は負のエラー値を返します。

rpmsg_trysendto()

説明

リモート プロセッサにメッセージを送信します。この関数は、リモート dst アドレスに長さ len のデータを送信します。メッセージは、ept のソース アドレスを使用して、ept チャンネルが所属するリモート プロセッサに送信されます。

利用可能な TX バッファがない場合、この関数はいずれかのバッファが利用可能になるまで待たずに、直ちに -ENOMEM を返します。

用途

```
static inline int rpmsg_trysendto(struct rpmsg_endpoint *ept, const void *data,
                                  int len, uint32_t dst)
```

引数

ept	RPMsg エンドポイント
data	メッセージのペイロード
len	ペイロードの長さ
dst	デスティネーション アドレス

戻り値

送信したバイト数を返し、失敗した場合は負のエラー値を返します。

rpmsg_trysend_offchannel()

説明

明示的な src/dst アドレスを使用してメッセージを送信します。この関数は、長さ len のデータをリモート dst アドレスに送信し、ソース アドレスとして src を使用します。メッセージは、ept チャンネルが所属するリモート プロセッサに送信されます。

利用可能な TX バッファがない場合、この関数はいずれかのバッファが利用可能になるまで待たずに、直ちに -ENOMEM を返します。

用途

```
static inline int rpmsg_trysend_offchannel(struct rpmsg_endpoint *ept,
                                           uint32_t src, uint32_t dst,
                                           const void *data, int len)
```

引数

<code>ept</code>	RPMsg エンドポイント
<code>src</code>	ソース アドレス
<code>dst</code>	デスティネーション アドレス
<code>data</code>	メッセージのペイロード
<code>len</code>	ペイロードの長さ

戻り値

送信したバイト数を返し、失敗した場合は負のエラー値を返します。

rpmsg_init_ept

説明

RPMsg エンドポイントを初期化します。名前、ソースアドレス、`remoteproc` アドレス、エンドポイント コールバック、およびエンドポイント破棄コールバックを指定して、RPMsg エンドポイントを初期化します。

用途

```
static inline void rpmsg_init_ept(struct rpmsg_endpoint *ept,
    const char *name,
    uint32_t src, uint32_t dest,
    rpmsg_ept_cb cb,
    rpmsg_ns_unbind_cb ns_unbind_cb)
```

引数

<code>ept</code>	RPMsg エンドポイントへのポインター
<code>name</code>	エンドポイントに関連付けられるサービス名
<code>src</code>	エンドポイントのローカル アドレス
<code>dest</code>	エンドポイントのターゲット アドレス
<code>cb</code>	エンドポイント コールバック
<code>ns_unbind_cb</code>	リモート <code>ept</code> の破棄時に呼び出される、エンドポイント サービス アンバインド コールバック

rpmsg_create_ept

説明

RPMsg エンドポイントを作成し、RPMsg デバイスに登録します。名前、ソース アドレス、remoteproc アドレス、エンドポイント コールバック、およびエンドポイント破棄コールバックを指定して RPMsg エンドポイントを作成し、RPMsg デバイスに登録します。基本的に、RPMsg エンドポイントは RPMsg アドレスと rx コールバック ハンドラーをバインドするものであり、RPMsg バス上のリスナーを表します。

RPMsg クライアントは、リモートと対話するためのエンドポイントを作成する必要があります。RPMsg クライアントは、少なくともチャンネル名とメッセージ通知用コールバックを提供する必要があります。デフォルトでは、エンドポイントのソース アドレスは RPMSG_ADDR_ANY に設定されます。

オプションとして、一部の RPMsg クライアントは、特定のソース アドレスを持つエンドポイントを指定できます。

用途

```
int rpmsg_create_ept(struct rpmsg_endpoint *ept, struct rpmsg_device *rdev,
                    const char *name, uint32_t src, uint32_t dest,
                    rpmsg_ept_cb cb, rpmsg_ns_unbind_cb ns_unbind_cb)
```

引数

ept	RPMsg エンドポイントへのポインタ
name	エンドポイントに関連付けられるサービス名
src	エンドポイントのローカル アドレス
dest	エンドポイントのターゲット アドレス
cb	エンドポイント コールバック
ns_unbind_cb	リモート ept の破棄時に呼び出される、エンドポイント サービス アンバインド コールバック

rpmsg_destroy_ept

説明

RPMsg エンドポイントを破棄し、RPMsg デバイスへの登録を解除します。RPMsg デバイスへの RPMsg エンドポイントの登録を解除し、エンドポイント破棄コールバック (提供されている場合) を呼び出します。

用途

```
void rpmsg_destroy_ept(struct rpmsg_endpoint *ept);
```

引数

ept - RPMsg エンドポイントへのポインタ

is_rpmsg_ept_ready

説明

RPMsg エンドポイントを送信できるかどうかをチェックします。

用途

```
static inline unsigned int is_rpmsg_ept_ready(struct rpmsg_endpoint *ept)
```

引数

ept - RPMsg エンドポイントへのポインター

戻り値

RPMsg エンドポイントのローカル アドレスとデスティネーション アドレスの両方が設定されている場合は 1、そうでない場合は 0。

rpmsg_virtio_get_buffer_size

説明

RPMsg virtio バッファ サイズを取得します。

用途

```
int rpmsg_virtio_get_buffer_size(struct rpmsg_device *rdev);
```

引数

rdev - RPMsg デバイスへのポインター

戻り値

文字が返された場合は次の利用可能なバッファ サイズの値、失敗した場合は負の値。

rpmsg_init_vdev

説明

RPMMsg virtio デバイスを初期化します。

マスター側: RPMMsg virtio キューと共有バッファを初期化します。shm のアドレスを ANY にすることができます。この場合、関数は、システムの共有メモリ プールから共有メモリを取得します。vdev が RPMMsg ネーム サービス機能を備えている場合、この API はネーム サービスのエンドポイントを作成します。

スレーブ側: マスター側でドライバー レディが設定されるまで、この API は値を返しません。

用途

```
int rpmsg_init_vdev(struct rpmsg_virtio_device *rvdev,
                   struct virtio_device *vdev,
                   rpmsg_ns_bind_cb ns_bind_cb,
                   struct metal_io_region *shm_io,
                   struct rpmsg_virtio_shm_pool *shpool);
```

引数

rvdev	RPMMsg virtio エンドポイントへのポインター
vdev	virtio デバイスへのポインター
ns_bind_cb	ローカルエンドポイントがバインドされるのを待たずにネームサービスをアナウンスするコールバック ハンドラー
shm_io	共有メモリ I/O 領域へのポインター
shpool	共有メモリ プールへのポインター。この構造体を埋めるには、RPMMsg_virtio_init_shm_pool を先に呼び出す必要があります。

戻り値

関数選択のステータス。

rpmsg_deinit_vdev

説明

RPMMsg virtio デバイスの初期化を解除します。

用途

```
void rpmsg_deinit_vdev(struct rpmsg_virtio_device *rvdev);
```

引数

rdev - RPMMsg virtio デバイスへのポインター

rpmsg_virtio_init_shm_pool

説明

RPMMsg virtio が備えているデフォルトの共有バッファプール インプリメンテーションを初期化します。このプールに割り当てられるメモリは、RPMMsg virtio 専用です。rpmsg_init_vdev を呼び出す前にこの関数を呼び出して、rpmsg_virtio_shm_pool 構造体を初期化する必要があります。

用途

```
void rpmsg_virtio_init_shm_pool(struct rpmsg_virtio_shm_pool *shpool,  
                               void *shbuf, size_t size);
```

引数

shpool	共有バッファプール構造体へのポインター
shbuf	共有バッファの開始点へのポインター
size	共有バッファ全体のサイズ

rpmsg_virtio_get_rpmsg_device

説明

この関数は、RPMMsg virtio デバイスから RPMMsg デバイスを取得します。

用途

```
static inline struct rpmsg_device *  
rpmsg_virtio_get_rpmsg_device(struct rpmsg_virtio_device *rvdev)
```

引数

rdev - RPMMsg virtio デバイスへのポインター

戻り値

RPMMsg virtio デバイスによって指示される RPMMsg デバイス。

rpmsg_virtio_shm_pool_get_buffer

説明

この関数は、共有メモリ プール内のバッファを取得します。

RPMMsg virtio は、デフォルトの共有バッファプール インプリメンテーションを備えています。このプールに割り当てられるメモリは、RPMMsg virtio 専用です。共有バッファのアロケーションを変更する場合は、独自の rpmsg_virtio_shm_pool_get_buffer 関数をインプリメントできます。

用途

```
metal_weak void *  
rpmsg_virtio_shm_pool_get_buffer(struct rpmsg_virtio_shm_pool *shpool,  
                                size_t size);
```

引数

shpool	共有バッファプールへのポインター
size	共有バッファ全体のサイズ

戻り値

空きバッファが利用可能な場合はバッファへのポインター、そうでない場合は NULL。

その他のリソースおよび法的通知

ザイリンクス リソース

アンサー、資料、ダウンロード、フォーラムなどのサポート リソースは、[ザイリンクス サポート サイト](#)を参照してください。

ソリューション センター

デバイス、ツール、IP のサポートについては、[ザイリンクス ソリューション センター](#)を参照してください。デザイン アシスタント、デザイン アドバイザリ、トラブルシューティングのヒントなどが含まれます。

Documentation Navigator およびデザイン ハブ

ザイリンクス Documentation Navigator (DocNav) では、ザイリンクスの資料、ビデオ、サポート リソースにアクセスでき、特定の情報を取得するためにフィルター機能や検索機能を利用できます。DocNav を開くには、次のいずれかを実行します。

- Vivado IDE で [Help] → [Documentation and Tutorials] をクリックします。
- Windows で [スタート] → [すべてのプログラム] → [Xilinx Design Tools] → [DocNav] をクリックします。
- Linux コマンド プロンプトに「docnav」と入力します。

ザイリンクス デザイン ハブには、資料やビデオへのリンクがデザイン タスクおよびトピックごとにまとめられており、これらを参照することでキー コンセプトを学び、よくある質問 (FAQ) を参考に問題を解決できます。デザイン ハブにアクセスするには、次のいずれかを実行します。

- DocNav で [Design Hubs View] タブをクリックします。
- ザイリンクス ウェブサイトの[デザイン ハブ](#) ページを参照します。

注記: DocNav の詳細は、ザイリンクス ウェブサイトの [Documentation Navigator](#) ページを参照してください。



注意: DocNav からは、日本語版は参照できません。ウェブサイトのデザイン ハブ ページをご利用ください。

ザイリンクス資料

注記: 日本語版のバージョンは、英語版より古い場合があります。

1. OpenAMP Wiki: <http://www.wiki.xilinx.com/OpenAMP>
2. 『Zynq UltraScale+ MPSoC テクニカル リファレンス マニュアル』(UG1085: 英語版、日本語版)
3. 『ザイリンクス ソフトウェア開発キットのヘルプ』(UG782)
4. 『PetaLinux ツール資料: リファレンス ガイド』(UG1144: 英語版、日本語版)
5. 『Vitis 統合ソフトウェア プラットフォームの資料: エンベデッド ソフトウェア開発』(UG1400)
6. ザイリンクス libmetal ソース コード: <https://github.com/Xilinx/libmetal>
7. ザイリンクス OpenAMP ソース コード: <https://github.com/Xilinx/open-amp>

お読みください: 重要な法的通知

本通知に基づいて貴殿または貴社(本通知の被通知者が個人の場合には「貴殿」、法人その他の団体の場合には「貴社」。以下同じ)に開示される情報(以下「本情報」といいます)は、ザイリンクスの製品を選択および使用することのためにのみ提供されます。適用される法律が許容する最大限の範囲で、(1) 本情報は「現状有姿」、およびすべて受領者の責任で (with all faults) という状態で提供され、ザイリンクスは、本通知をもって、明示、黙示、法定を問わず(商品性、非侵害、特定目的適合性の保証を含みますがこれらに限られません)、すべての保証および条件を負わない(否認する)ものとします。また、(2) ザイリンクスは、本情報(貴殿または貴社による本情報の使用を含む)に関係し、起因し、関連する、いかなる種類・性質の損失または損害についても、責任を負わない(契約上、不法行為上(過失の場合を含む)、その他のいかなる責任の法理によるかを問わない)ものと、当該損失または損害には、直接、間接、特別、付随的、結果的な損失または損害(第三者が起こした行為の結果被った、データ、利益、業務上の信用の損失、その他あらゆる種類の損失や損害を含みます)が含まれるものとし、それは、たとえ当該損害や損失が合理的に予見可能であったり、ザイリンクスがそれらの可能性について助言を受けていた場合であったとしても同様です。ザイリンクスは、本情報に含まれるいかなる誤りも訂正する義務を負わず、本情報または製品仕様のアップデートを貴殿または貴社に知らせる義務も負いません。事前の書面による同意のない限り、貴殿または貴社は本情報を再生産、変更、頒布、または公に展示してはなりません。一定の製品は、ザイリンクスの限定的保証の諸条件に従うこととなるので、<https://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。IP コアは、ザイリンクスが貴殿または貴社に付与したライセンスに含まれる保証と補助的条件に従うことになります。ザイリンクスの製品は、フェイルセーフとして、または、フェイルセーフの動作を要求するアプリケーションに使用するために、設計されたり意図されたりしていません。そのような重大なアプリケーションにザイリンクスの製品を使用する場合のリスクと責任は、貴殿または貴社が単独で負うものです。<https://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。

自動車用のアプリケーションの免責条項

オートモーティブ製品(製品番号に「XA」が含まれる)は、ISO 26262 自動車用機能安全規格に従った安全コンセプトまたは余剰性の機能(「セーフティ設計」)がない限り、エアバッグの展開における使用または車両の制御に影響するアプリケーション(「セーフティ アプリケーション」)における使用は保証されていません。顧客は、製品を組み込むすべてのシステムについて、その使用前または提供前に安全を目的として十分なテストを行うものとします。セーフティ設計なしにセーフティ アプリケーションで製品を使用するリスクはすべて顧客が負い、製品の責任の制限を規定する適用法令および規則にのみ従うものとします。

© Copyright 2015-2020 Xilinx, Inc. Xilinx, Xilinx のロゴ、Alveo、Artix、Kintex、Spartan、Versal、Virtex、Vivado、Zynq、およびこの文書に含まれるその他の指定されたブランドは、米国およびその他の各国のザイリンクス社の商標です。AMBA、AMBA Designer、Arm、ARM1176JZ-S、CoreSight、Cortex、PrimeCell、Mali、および MPCore は、EU およびその他の各国の Arm 社の商標です。すべてのその他の商標は、それぞれの所有者に帰属します。

この資料に関するフィードバックおよびリンクなどの問題につきましては、jpn_trans_feedback@xilinx.com まで、または各ページの右下にある [フィードバック送信] ボタンをクリックすると表示されるフォームからお知らせください。フィードバックは日本語で入力可能です。いただきましたご意見を参考に早急に対応させていただきます。なお、このメール アドレスへのお問い合わせは受け付けておりません。あらかじめご了承ください。