参考資料

AN5289
アプリケーション･ノート

STM32WB シリーズ･マイクロコントローラで ワイヤレス･アプリケーションを構築する

概要

このアプリケーション･ノートは、STM32WB シリーズ･マイクロコントローラに基づく Bluetooth® Low Energy（BLE）または802.15.4規格の特定のアプリケーションを構築するために必要な手順を通じて、設計者を案内するために記載されています。この中には、非常に重要な情報がまとめられており、対応すべき側面が示されています。

このアプリケーション･ノートの情報を十分に活用してアプリケーションを開発するためには、STM32 マイクロコントローラ、Bluetooth LE 技術、802.15.4OpenThread プロトコル、および 802.15.4 MAC レイヤに精通している必要があり、低消費電力管理やタスク･シーケンシングなどのシステム･サービスを理解している必要があります。

参考資料

目次

参考資料

参考資料

参考資料

参考資料

参考資料

参考資料

表の一覧

参考資料

図の一覧

参考資料

参考資料

1 参照

[1]	UM2550(1)	Getting started with STM32CubeWB for STM32WB Series
[2]	RM0434 (1)	Multiprotocol wireless 32-bit MCU Arm®-based Cortex®-M4 with FPU, Bluetooth® Low-Energy and 802.15.4 radio solution
[3]	AN5270 (1)	STM32WBx5 Bluetooth® Low Energy (BLE) wireless interface
[4]	UM2442 (1)	Description of STM32WB HAL and low-layer drivers
[5]	UM2288 (1)	STM32CubeMonitor-RF software tool for wireless performance measurements
[6]	AN5185 (1)	ST firmware upgrade services for STM32WB Series
[7]	Bluetooth® 仕様	Bluetooth Core Specification (v4.0, v4.1, v4.2, v5.0)
[8]	MAC IEEE Std 802.15.4-2011	Specification of the 802_15_4 MAC standard
[9]	Thread 仕様	Thread specification V1.1 (Thread Group)

1. www.st.com から入手可能です。

参考資料

2 略記と略語のリスト

ACI	アプリケーション･コマンド･インタフェース
ATT	属性プロトコル
BLE	Bluetooth® Low Energy
CLI	コマンド･ライン･インタフェース
CoAP	コンストレインド･アプリケーション･プロトコル
CPU1	Cortex®-M4コア
CPU2	Cortex®-M0+ コア
D2D	デバイス間
DUT	試験対象デバイス
FUOTA	オーバー･ジ･エアー（無線）によるファームウェア更新
FUS	ファームウェアアップグレードサービス
GAP	汎用アクセス･プロファイル
GATT	汎用属性プロファイル
HCI	ホスト･コントローラ･インタフェース
L2CAP	論理リンク制御アダプテーション･レイヤ･プロトコル
LTK	ロングタームキー
OTA	オーバー･ジ･エアー（無線経由）
PDU	プロトコル･データ･ユニット
P2P	ピアツーピア
RFU	将来の使用のために予約済み。
SIG	専門部会
SM	セキュリティマネージャ
UUID	Universally unique identifier（汎用一意識別子）

3 ソフトウェアの概要

3.1 対応スタック

STM32WB シリーズマイクロコントローラは Arm®(a) コアをベースとしています。

ターゲットアプリケーションに基づいて、ロードする CPU2ファームウェアを選択します。

STM32WB シリーズマイクロコントローラのエコシステムは、図 1 に示す固有インタフェースを通じてアプリケーションによって制御される各種のスタック（表 1 参照）をサポートしています。

図 2 に示されているように、CPU2は BT HCI 標準インタフェースに対応しており、各種の Bluetooth LE スタックが CPU1上で動作可能です。

表 1. STM32WB シリーズマイクロコントローラがサポートしているスタック

サポートしているスタック	関連ファームウェア
Bluetooth LE	stm32wb5x_BLE_Stack_fw.bin stm32wb5x_BLE_HCILayer_fw.bin
Thread	stm32wb5x_Thread_FTD_fw.bin stm32wb5x_Thread_MTD_fw.bin
Bluetooth LE と Thread	stm32wb5x_BLE_Thread_fw.bin
MAC 802_15_4	stm32wb5x_Mac_802_15_4_fw.bin

a. Arm は、米国内およびその他の地域にある Arm Limited（またはその子会社）の登録商標です。

参考資料

図 1. STM32WB シリーズマイクロコントローラがサポートしているプロトコル

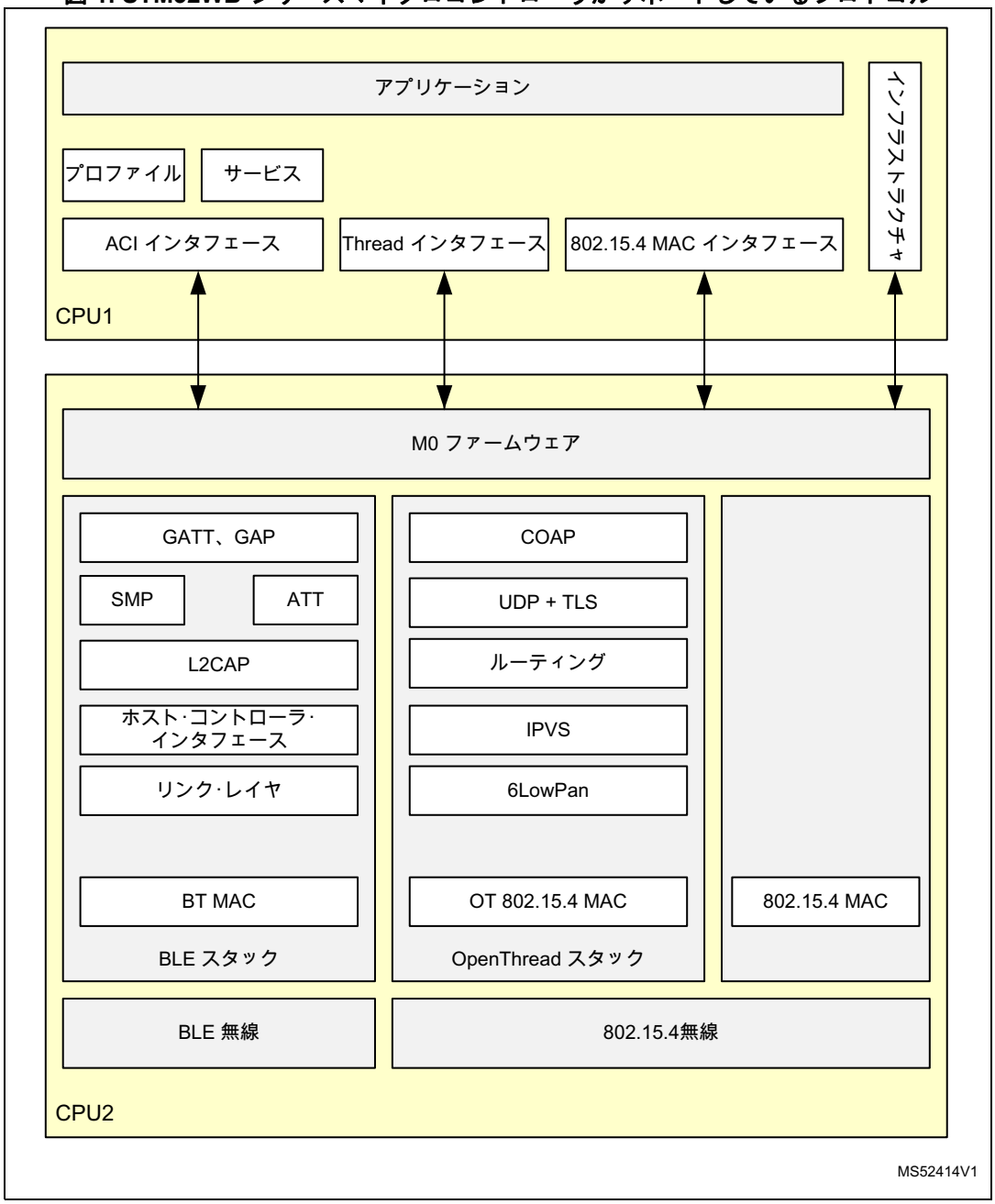
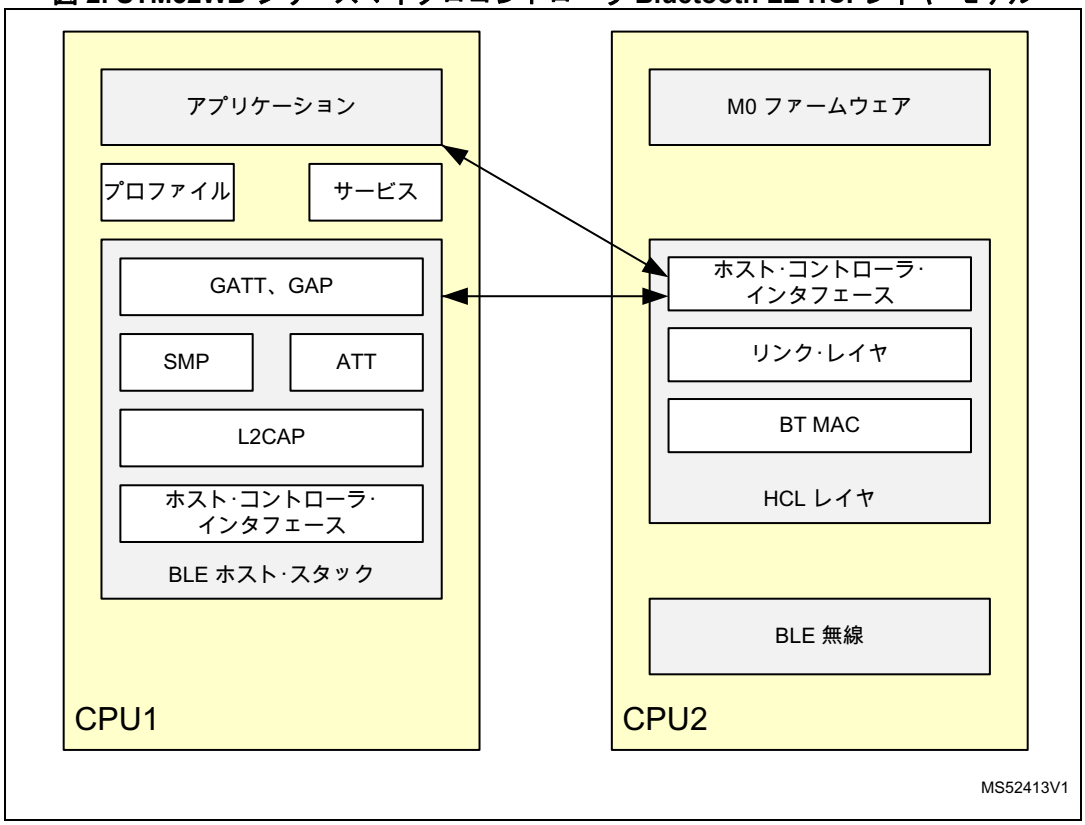


図 2. STM32WB シリーズマイクロコントローラ Bluetooth LE HCI レイヤ·モデル



3.2 Bluetooth LE アプリケーション

STM32WB アーキテクチャでは、アプリケーション CPU1 上で動作する Bluetooth LE プロファイルとアプリケーションを、Bluetooth LE ペリフェラルにあるリアルタイム性の面から分離します。

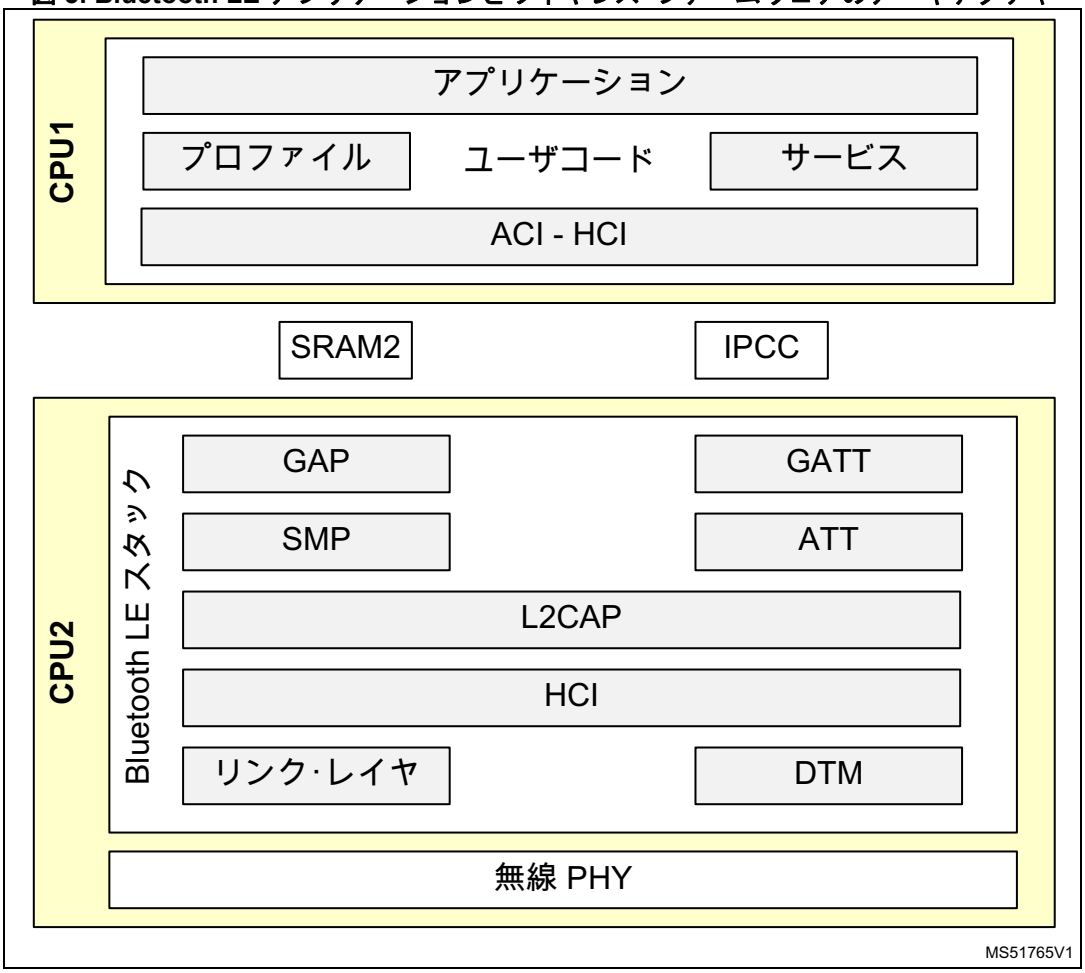
Bluetooth LE ペリフェラルには、GAP までのリンク·レイヤと GAP レイヤを処理するスタックが含まれている CPU2 プロセッサが組み込まれています。また、ここには物理的な 2.4GHz 無線も搭載されています。

アプリケーション CPU1 は、Bluetooth LE に送信するデータの収集と計算を行います。

CPU2には、以下のようなすべてのリアルタイム·リンク·レイヤと無線 PHY の相互作用の管理に必要である LE コントローラと LE ホストが含まれています。

- 低消費電力モードを管理する低消費電力マネージャ
- 動作に関する情報を出力するデバッグ·トレース
- Bluetooth LE スタック（LL、GAP、GATT）とインタフェースするメールボックス／IPCC

図 3. Bluetooth LE アプリケーションとワイヤレス·ファームウェアのアーキテクチャ



3.3 HCI レイヤ·インタフェース上への Bluetooth LE アプリケーションの構築

CPU2 は Bluetooth LE HCI レイヤのコプロセッサとして使用することができます。その場合、独自の HCI アプリケーションを実装するか、あるいは既存のオープンソース Bluetooth LE ホストスタックを使用することになります。

多くの Bluetooth LE ホストスタックでは、UART インタフェースを使用して Bluetooth LE HCI コプロセッサとの通信を行います。STM32WB シリーズマイクロコントローラの等価物理レイヤは、セクション 13.2：メールボックス·インタフェースに記載されているメールボックスです。

メールボックスは、Bluetooth LE チャネルとシステムチャネル両方へのインタフェースを備えています。Bluetooth LE ホストスタックは、メールボックスの Bluetooth LE チャネルに送信されるコマンドバッファの構築を 担うものであり、メールボックス経由で受信するイベントを報告するインタフェースを備えている必要があります。メールボックスに対する Bluetooth LE ホストスタックの適応に加えて、ユーザは非同期パケットのリリースが可能になったときにはメールボックス·ドライバに通知を行う必要があります。

システムチャネルは Bluetooth LE ホストスタックによる操作は行われません。カスタムのトランスポート·レイヤを実装し、メールボックス·ドライバに送信されるシステムコマンドバッファを構築して、メールボックスから受信するイベント（メールボックス·ドライバに対する非同期バッファのリ

リース通知を含む）を管理する必要があります。あるいは、システムコマンドバッファの構築を担うトランスポート·レイヤの上にインタフェースを提供するメールボックス拡張ドライバ（セクション 13.3：メールボックス·インタフェース - 拡張に記載）を使用して、システム非同期イベントを管理することもできます。

セクション 11.2：Thread_Coap_DataTransferに記載されているようにメールボックスを使用してBluetooth LE HCI レイヤのコプロセッサの上にアプリケーションを構築する例として、BLE_TransparentMode プロジェクトを使用することもできます。

3.4 Thread アプリケーション

OpenThread スタックは CPU2コア上で動作し、完全な Thread アプリケーションを構築するためのAPI のセットを CPU1側にエクスポートします。次の3つの CPU2 ファームウェアが Thread プロトコルをサポートしています。

- **sm32wb5x_Thread_FTD_fw**：この場合、デバイスは境界ルータ（リーダ、ルータ、エンドデバイス、スリーピーエンドデバイスなど）を除くすべての Thread ロールをサポートしています。
- **stm32wb5x_Thread_MTD_fw**：この場合、デバイスはエンドデバイスまたはスリーピーエンドデバイスとしてのみ振る舞うことができます。この構成では、FTD 構成よりもメモリ空間がセーブされます。
- **stm32wb5x_BLE_Thread_fw**：この場合、デバイスは Thread（FTD）と静的同時モードのBluetooth LE の両方をサポートしています（詳細はセクション 3.6：参照）。

3.5 MAC 802_15_4 アプリケーション

STM32wb5x_Mac_802_15_4_fw CPU2 ファームウェアをダウンロードすると、CPU1は 802_15_4 MAC レイヤに直接アクセスして、その上に独自アプリケーションを構築できるようになります。

3.6 Bluetooth LE と Thread アプリケーションの同時使用

STM32WB シリーズマイクロコントローラは、「静的同時モード」（スイッチモードとも呼ばれます）をサポートしています。

www.st.comから入手可能な stm32wb5x_BLE_Thread_fw CPU2ファームウェアには、（Bluetooth LE と Thread の）どちらのスタックも組み込まれています。あるプロトコルから別のプロトコルへの切替えは、システム·アプリケーション·コマンドを通じて行われます。このモードでは、動作中のプロトコルは、別のプロトコルをアクティブにする前にシステムによって無効にされます。STM32WB デバイスは、Bluetooth LE スタックが完全に停止した後に Bluetooth LE から Thread に切り替わります（その反対も同様）。ネットワークを毎回再接続する必要があるため、これらの遷移には数秒かかります。

4 STM32WB ソフトウェアアーキテクチャ

4.1 主要原理

- CPU2上で動作するすべてのコードは、暗号化バイナリとして提供されます。
- ユーザからはブラックボックスとなります。
- CPU1上で動作するすべてのコードは、ソースコードとして提供されます。
- CPU 間の通信は「メールボックス」を経由して行われます。

標準の STM32Cube 提供パッケージには、次のような STM32WB リソースが含まれています。

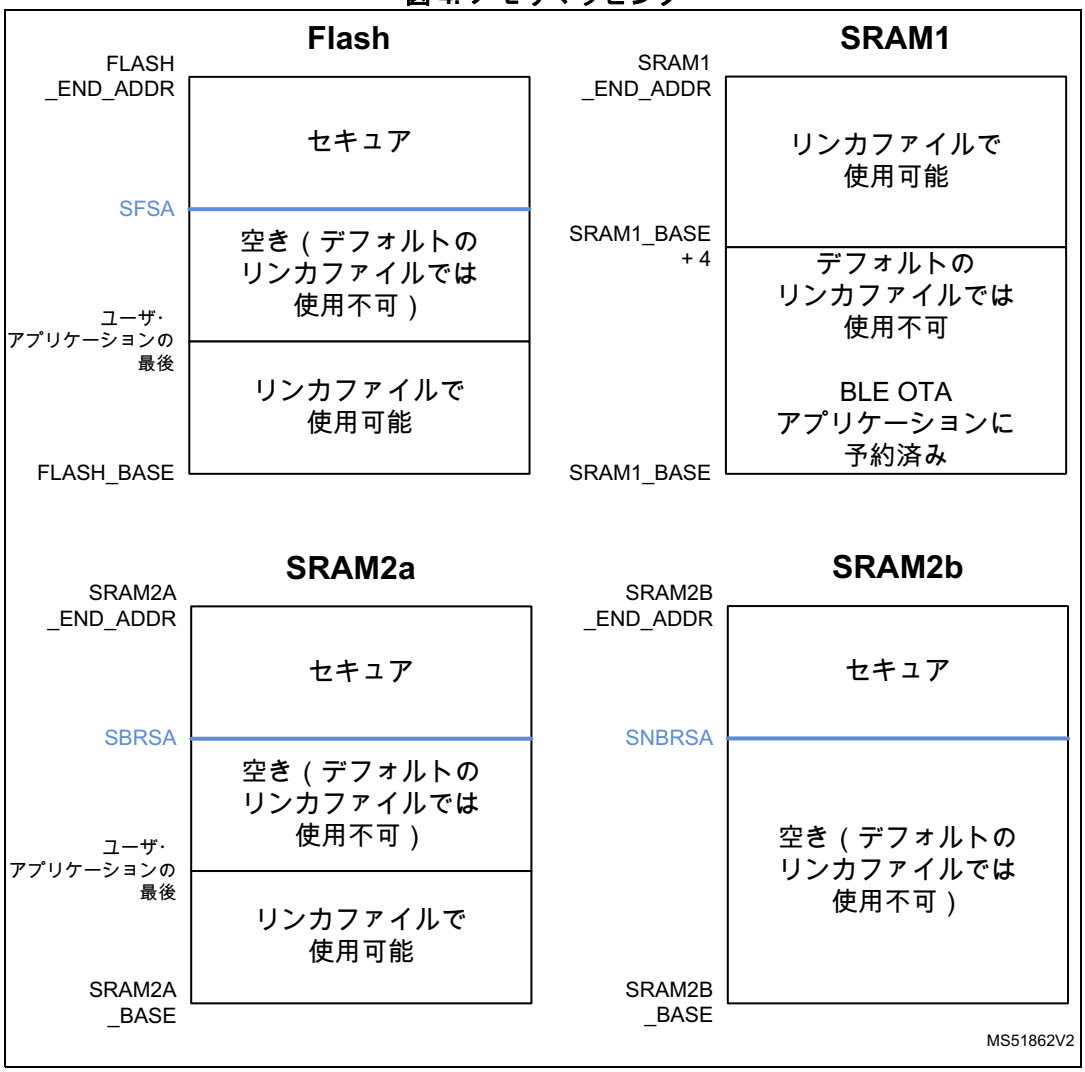
- ハードウェアレジスタにアクセスするための HAL/LL
- BSP
- ミドルウェア（FreeRTOS、USB デバイスなど）

さらに、以下のアプリケーションによって効率的なシステムインテグレーションが可能です。

- タスクをバックグラウンドで実行し、動作を行わない場合には低消費電力モードに移行するシーケンサ
- （STOP モードと STANDBY モードにおいて）RTC 上で動作する仮想タイマをアプリケーションに提供するタイマ･サーバ

4.2 メモリマッピング

図 4. メモリマッピング



Flash、SRAM2a、SRAM2b メモリには、CPU1からは読出しも書込みも行えない セキュア·セクションが含まれています。各メモリのセキュア開始アドレスは、図 4 で青色で示されているオプションバイトから読み出すことができます。

- Flash メモリは SFSA
- SRAM2a は SBRSA（STANDBY で保持）
- SRAM2b は SNBRSA

これらのオプションバイトは、CPU2上で動作する FUS のみによって書き込まれます。これは、FUS によってインストールされる CPU2 更新のたびに行われます。

ユーザアプリケーションは、RF スタックのバージョンが異なると、使用可能なメモリも異なる可能性があることを考慮する必要があります。ユーザアプリケーションが使用可能な空間は、STM32WB コプロセッサ·ワイヤレス·バイナリのリリースノートから取得可能です。RF スタックのインストー

ル·アドレスもユーザ Flash メモリ領域の境界アドレスとなります。製品耐用年数の間の更新をサポートするために、一定のマージンが CPU2 ドメインに確実に含まれるようにしてください。

境界の単位は、Flash メモリは 4KB、SRAM2a と SRAM2b はどちらも 1KB です。

リンカファイルは、提供されたすべての Bluetooth LE／Thread アプリケーション（BLE_Thread_Static、BLE_HeartRate_ota、BLE_p2pServer_ota を除く）で同じものとなります。使用可能なメモリは、提供されたすべてのアプリケーションに適するように選ばれています。CPU2が要求するメモリが小さい Bluetooth LE のようなアプリケーションでは、リンカファイルを更新してアプリケーションに割り当てるメモリを増加させることができます。

専用アプリケーションに対して使用可能なメモリを最適化するには、次のガイドラインに沿ってリンカファイルを更新する必要があります。

- Flash メモリ : 使用可能メモリの終了アドレスは、最大で SFSA アドレスまで動かすことが可能です。CPU2 更新が必要である場合には、新しい暗号化 CPU2 FW 更新をアップロードするために、セキュアメモリのすぐ下に十分な空きメモリが存在する必要があります。必要なメモリサイズは、更新する CPU2 FW（Bluetooth LE、Thread、同時 Bluetooth LE／Thread）に依存します（[1] 参照）。
- SRAM1 : リンカファイルで使用不可となっている最初の 32ビットが必要となるのは、BLE_OTA アプリケーションのみです。それ以外のすべてのアプリケーションでは、開始アドレスを SRAM1_BASE + 4 から SRAM1_BASE に動かすことができます。
- SRAM2a : 使用可能メモリの終了アドレスは、最大で SBRSA アドレスまで動かすことが可能です。CPU2 更新のサポートが必要である場合には、より多くのセクタがセキュアであることを必要とする新しい CPU2 FW 更新をサポートするために、セキュアメモリのすぐ下にいくつかの空きセクタが存在する必要があります。
- SRAM2b : Thread プロトコルをサポートするあらゆる FW CPU2 に対してすべてセキュアであることから、SRAM2b はリンカファイルの一部ではありません。Bluetooth LE のみのアプリケーションでは、RW データを SRAM2B_BASE から SNBRSA アドレスまでの SRAM2B にマッピングする新しいセクションでリンカファイルを更新しても構いません。CPU2 更新のサポートが必要である場合には、より多くのセクタがセキュアであることを必要とする新しい CPU2 FW 更新をサポートするために、セキュアメモリのすぐ下にいくつかの空きセクタが存在する必要があります。

RF がアクティブである場合、STOP2 がサポートされる最も低消費電力なモードとなります。ユーザアプリケーションが STANDBY モードに移行する必要がある場合、最初にすべての RF 動作を停止し、STANDBY モードを抜けるときに CPU2 を再初期化する必要があります。ユーザアプリケーションは、非セキュア SRAM2a 全体を使用してそれ自体の（STANDBY モードで保持される必要のある）データを格納できます。

4.3 共有ペリフェラル

どちらの CPU からも同時にアクセス可能なすべてのペリフェラルは、ハードウェアセマフォによって保護されています。これらのペリフェラルにアクセスする前には、最初に関連するセマフォを取得して、その後に解放する必要があります。

表 2. セマフォ

セマフォ	目的
Sem0	RNG - 全レジスタ
Sem1	PKA - 全レジスタ
Sem2	FLASH - 全レジスタ
Sem3	RCC_CR RCC_EXTCFGR RCC_CFGR RCC_SMPSCR
Sem4	STOP モード実装用のクロック制御メカニズム
Sem5	RCC_CRRCR RCC_CCIPR
Sem6	CPU2 が Flash メモリの中のデータを書込み／消去することを防止するために CPU1 が使用
Sem7	CPU1 が Flash メモリの中のデータを書込み／消去することを防止するために CPU2 が使用

内部タスク制御のためにアプリケーションがセマフォを使用する必要がある場合には、新機能が追加されて別のセマフォが必要となる可能性のある、CPU1 における将来のワイヤレス･ファームウェア更新と互換性があるように、Sem31 から下方向に使い始めることを推奨します。

Sem0 は、2つの CPU 間で RNG IP を共有するために使用します。セマフォは、生成する必要のある RNG 数と RNG ソースクロック速度に応じた時間間隔で CPU2 によって取得されます。RNG 数を取得するための遅延を緩和するために、起動時に RNG 数のプールを作成し、アプリケーションによっていくつかの数が取得された際には低プライオリティタスクでそのプールをフィルして、フル状態に保つことを推奨します。Sem0 の使用方法を図 8 に示します。

Sem 0は、USB ユースケースでも使用できます。それ以上 USB を使用せず、アプリケーションによってオフにする必要がある場合には、CLK48クロックをオフにする前に Sem 0を取得する必要があります。これが必要であるのは USB と RNG が同一クロックを共有していて、CPU1 が USB をオフにする必要がある同じ時間に CPU2 が RNG を使用できるようにするためです（図 9 を参照）。

Sem1 は、2つの CPU 間で PKA IP を共有するために使用します。

Sem2 は、2つの CPU 間で FLASH IP を共有するために使用します。セマフォは、Flash メモリに書き込むデータ数と消去するセクタ数に応じた時間間隔で CPU2 によって取得されます。Bluetooth LE スタックは、ペアリング情報（ボンディング有効時）と GATT 属性キャッシュを Flash メモリに書き込みます。

Sem3 は、低消費電力管理に使用されます。Bluetooth LE RF の動作中は、CPU1によって 500µs を超えてロックされてはなりません。アルゴリズムについては、図 6 および図 7 で詳細に説明されています。

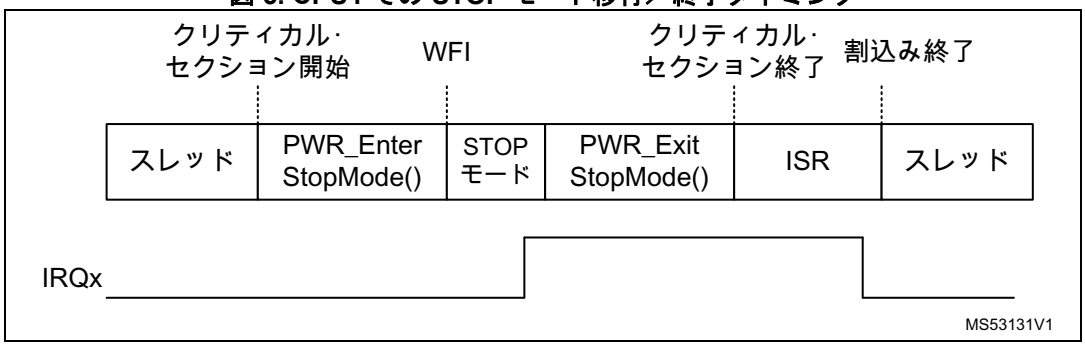
Sem4 は、1つの CPU が低消費電力モードを終了してもう 1つの CPU が低消費電力モードに移行する場合に、システムクロックの切替えにおける競合状態を処理するために使用されます。アルゴリズムについては、図 6 および図 7 で詳細に説明されています。

Sem3 と Sem4 は、STOP モードとの移行／終了の例で使用されています。

STOP モードからのウェイクアップの前後に、図 6 および図 7 で示したアルゴリズムが確実に実行されるようにする必要があります。これらのルーチン（図 5 参照）は、シーケンサまたは RTOS の IDLE タスクの中に実装されるのが一般的です。この実装は、WFI がクリティカル·セクションからコールされた場合に、MCU は割込みリクエストによってウェイクアップするものの、ISR を実行する代わりに WFI の後の次の命令を継続して実行するという事実を利用しています。クリティカル·セクションを終了した後になって初めて、ISR は実行されます。

```
PRIMASK = 1; // すべての割込みをマスク（クリティカル·セクションに移行）
PWR_EnterStopMode()
WFI
PWR_ExitStopMode()
PRIMASK = 0; // すべての割込みをマスク解除（クリティカル·セクションを終了）
```

図 5. CPU1 での STOP モード移行／終了タイミング



参考資料

図 6. CPU1 で STOP モードに移行するアルゴリズム

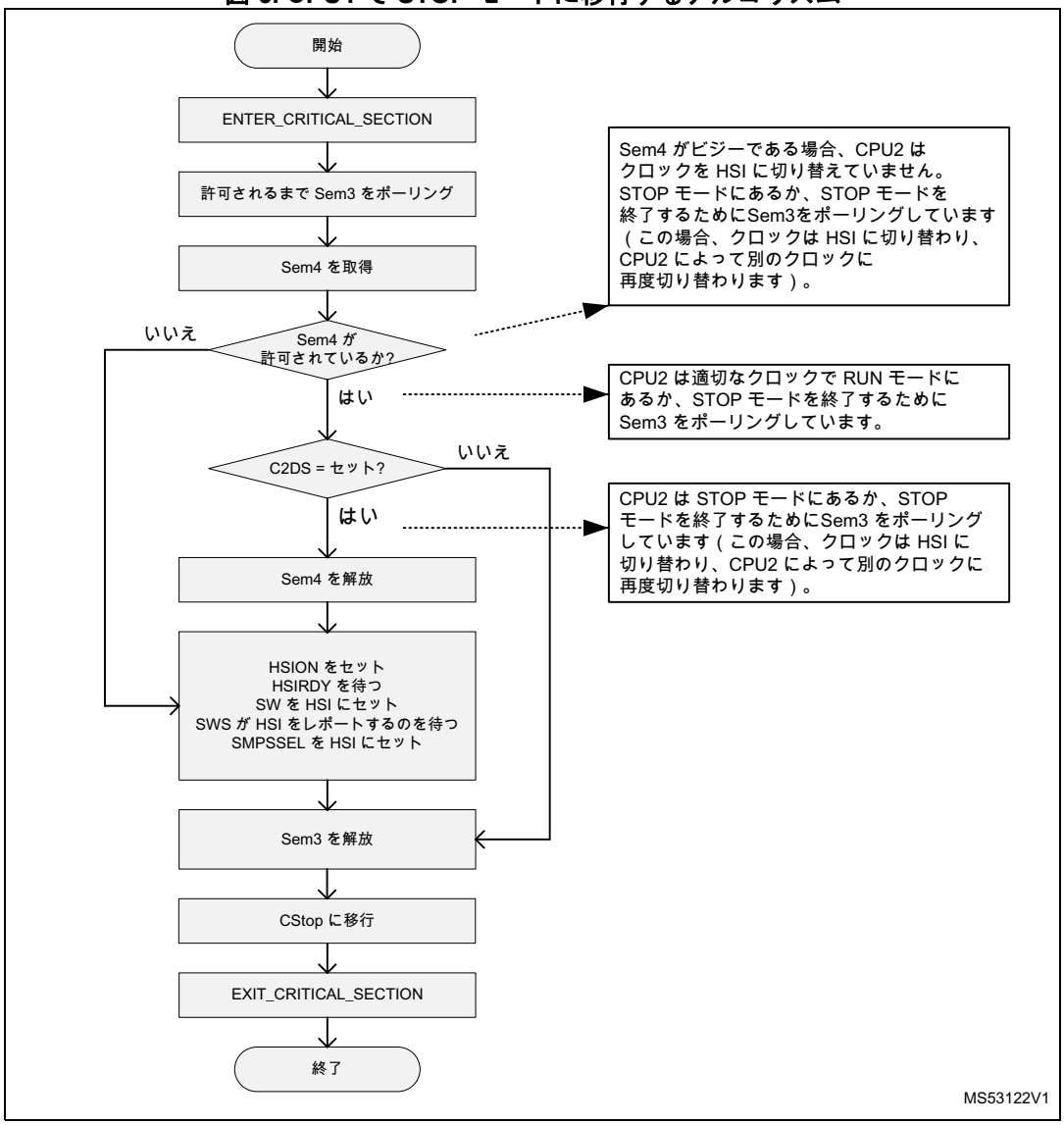
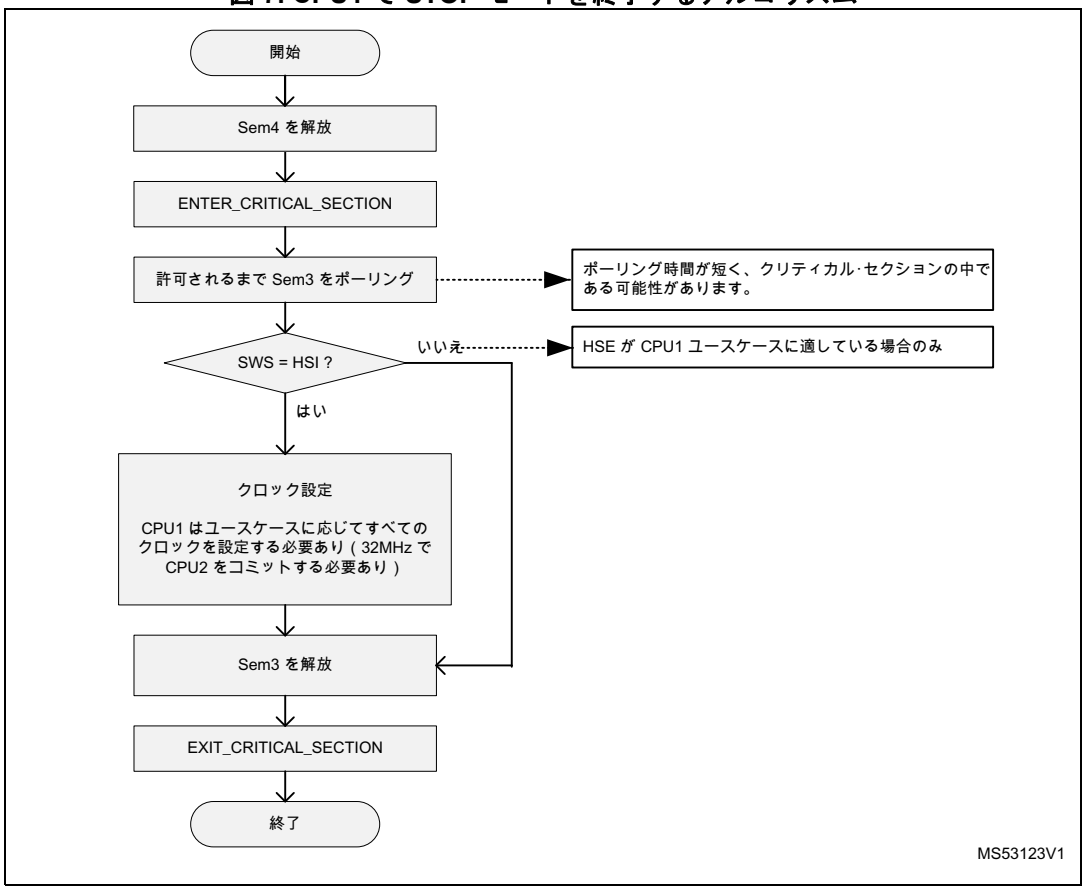


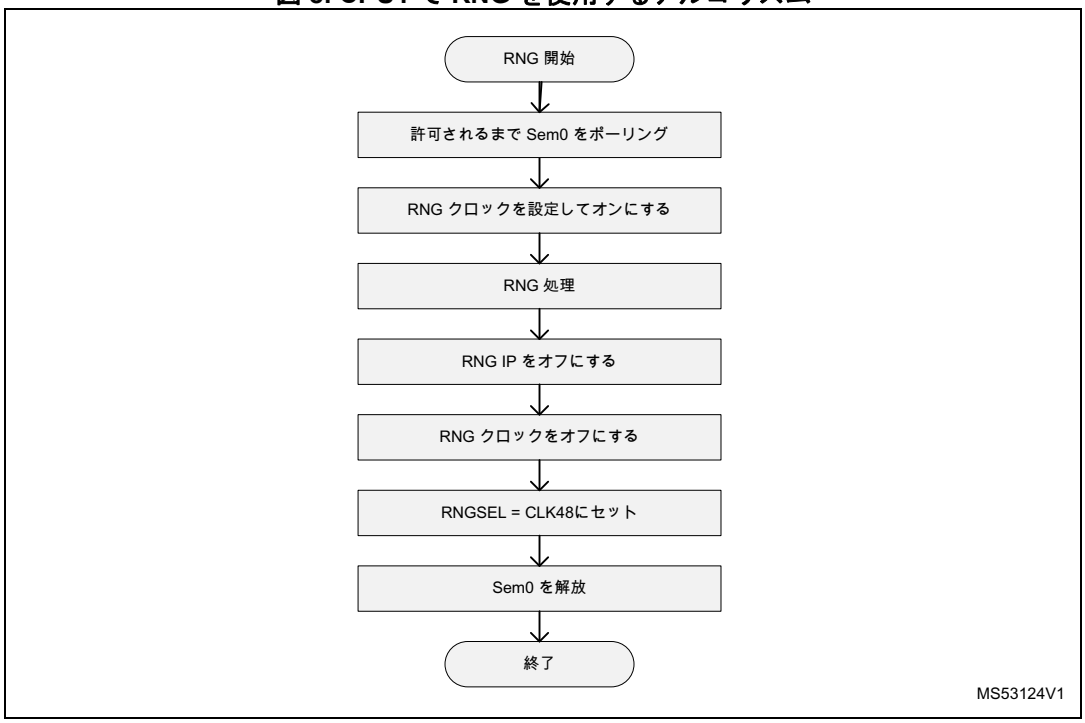
図 7. CPU1 で STOP モードを終了するアルゴリズム



Sem5 は、RNG/USB CLK48 ソースクロックを制御するために使用します。CPU2 は、RNG IP（Sem0）が使用されている場合のみクロックを更新したりオフにしたりします。

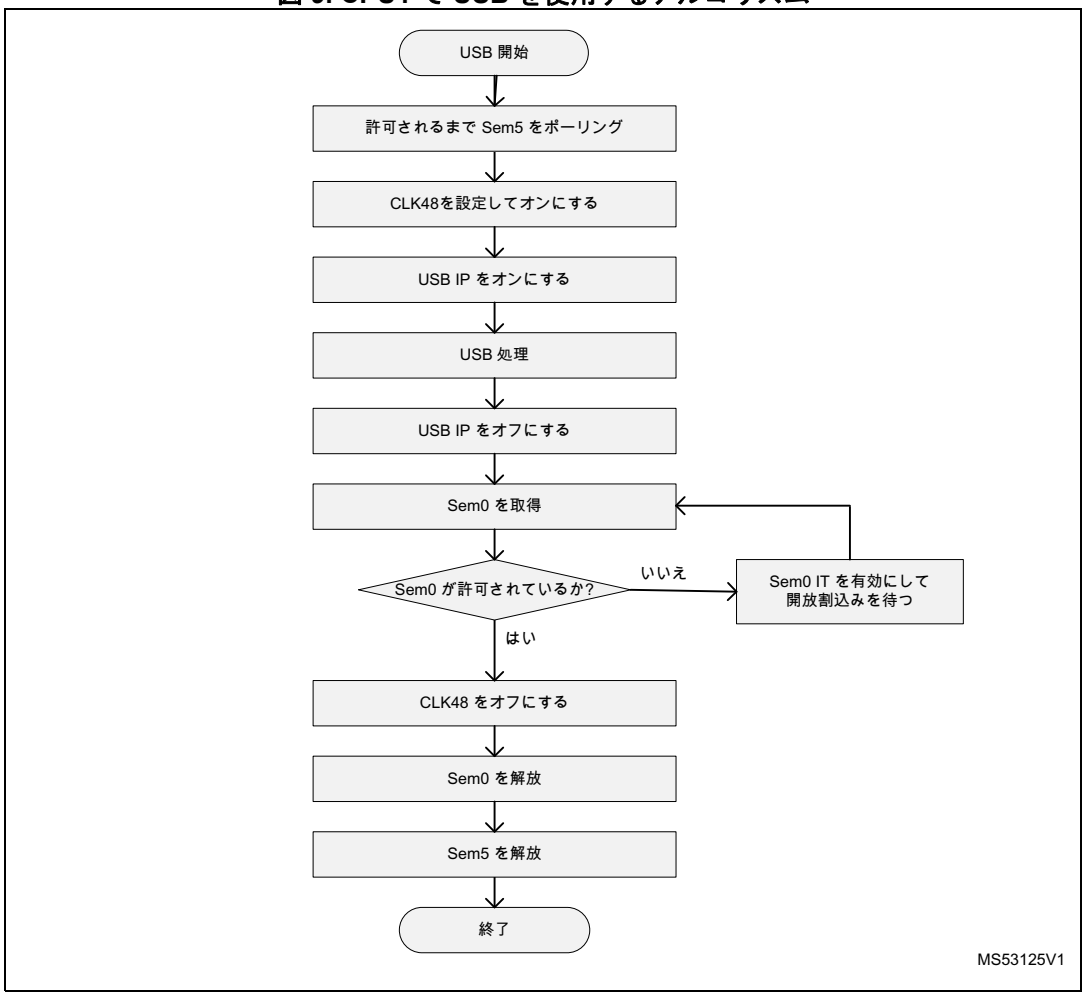
CPU2 との競合状態を回避するため、CPU1 がクロックをオフにする必要があるときには、RNG IP を使用していない場合であっても必ず最初に Sem0 を取得する必要があります。このメカニズムを Bluetooth LE P-NUCLEO-WB55.USB ドングルの例に示します（セクション 7.1.3: 設定および図 9 参照）。これは、CPU2 には影響を及ぼしません。

図 8. CPU1 で RNG を使用するアルゴリズム



注： Sem5は、CPU2が最初に Sem0を取得せずに取得されることはないため取得されません。このアルゴリズムは、RNG クロックソースの設定前に Sem5 を取得するように更新可能です。

図 9. CPU1 で USB を使用するアルゴリズム



USB IP と RNG IP は同じソースクロックを共有します。クロックをオフにする前に、USB ドライバは CPU2 がクロックを必要としているかどうかを最初にチェックする必要があります。CPU2との競合状態を回避するため、CPU1はクロックをオフする前に、最初に Sem0（RNG セマフォ、CPU2 は USB を不使用）を取得する必要があります。

Sem0 がビジー状態である場合、CPU1 はクロックをオフするために Sem0 が解放されるのを待つ必要があります。これが必要であるのは、CPU1 が USB を解放するのと同時に CPU2 が RNG を解放して、オシレータがオンのままとなる競合状態が存在する可能性があるためです。

Sem6 は、CPU2 によってリクエストされた書込み／消去操作に対して、CPU1 のタイミングを保護するために使用します。CPU1 は、Sem6 を取得して、CPU2 または CPU1 の他の処理が Flash メモリの中のデータを書き込んだり消去したりすることを防止するものとします。CPU1 がセマフォを保持可能な時間に対する制限はありませんが、セマフォが取得されている限り、CPU2 はペアリング情報もクライアント·ディスクリプタ情報もメモリに書き込むことはできません。

CPU1は、書込みまたは消去の操作を完了するために必要な時間の間ストールする余裕がある場合に限って Sem6 を解放する必要があります。

CPU1 と同様に、CPU2 は図 10 に記載されているアルゴリズムを実行します。Flash メモリにデータの書込みか消去を行う前に Sem6 の取得を試行し、成功した場合には、データの書込み／消去を行っ

てからセマフォを解放します。CPU1 が自分のタイミングを保護する必要がある場合、取得できるまで Sem6 をポーリングします。

Sem7 は、CPU1 によってリクエストされた Flash メモリとの書込み／消去操作に対して、CPU2 のタイミングを保護するために使用します。CPU1は、書込みまたは消去を行う前に Sem7を取得する必要があります。Sem7 は、1回の書込みまたは消去の操作のたびに取得と解放を行う必要がありますが、書込み／消去のタイミングから 0.5ms を超えてはならないものとします。この要件への適合のために、コードはクリティカル·セクションで実行する必要があります。アルゴリズムについては、図 10 で説明されています。

4.4 シーケンサ

シーケンサは登録された関数を 1つずつ実行します。以下の機能を備えています。

- 最大 32関数までサポート
- 実行する関数をリクエスト
- 関数の実行を有効化／無効化
- イベント受信に基づいてブロッキング·インタフェースを提供

シーケンサは、シンプルなバックグラウンド·スケジューリング機能を備えています。シーケンサに保留中の実行待ちタスクが全くない場合に、セキュアな方法の低消費電力モード（イベント喪失なし）を実行するフックを提供します。アプリケーションが次に進む前に特定のイベントを待つ効率的メカニズムも提供されます。シーケンサがある特定のイベントを待っている状態においては、アプリケーションが低消費電力モードに移行することも、別のコードを実行することも可能であるフックを提供します。

4.4.1 実装

シーケンサを使用するには、アプリケーションは以下の手順を実行する必要があります。

- サポートする関数の最大数をセットします（UTIL_SEQ_CONF_TASK_NBR に対して値を定義することで行われます）。
- UTIL_SEQ_RegTask() を用いてシーケンサがサポートする関数を登録します。
- UTIL_SEQ_Run() をコールしてシーケンサを起動し、バックグラウンド while ループを実行します。
- 関数を実行する必要があるときに UTIL_SEQ_SetTask() をコールします。

4.4.2 インタフェース

表 3. インタフェース関数

機能	説明
`void UTIL_SEQ_Idle( void );`	実行すべきものが存在しない場合に（クリティカル·セクションの中で -PRIMASK）コールします。
`void UTIL_SEQ_Run( UTIL_SEQ_bm_t mask_bm )`	保留されていてマスク mask_bm で有効化されている関数の実行をシーケンサにリクエストします。
`void UTIL_SEQ_RegTask(UTIL_SEQ_bm_t task_id_bm, uint32_t flags, void (*task)( void ))`	シグナル（task_id_bm）に関連付けられている関数（task）をシーケンサに登録します。task_id_bm にセットされているビットは 1つだけである必要があります。
`void UTIL_SEQ_SetTask( UTIL_SEQ_bm_t task_id_bm,`	task_id_bm に関連付けられている関数の実行をリクエストします。 task_prio は、関数が終了している場合のみシーケンサによって評価されます。同時に複数の関数が保留されている場合、優先順位が最も高い（0）ものが実行されます。
`void UTIL_SEQ_PauseTask( UTIL_SEQ_bm_t task_id_bm )`	シーケンサによる task_id_bm に関連付けられている関数の実行を無効にします。
`void UTIL_SEQ_ResumeTask( UTIL_SEQ_bm_t task_id_bm )`	シーケンサによる task_id_bm に関連付けられている関数の実行を有効にします。
`void UTIL_SEQ_WaitEvt( UTIL_SEQ_bm_t evt_id_bm )`	特定イベント event evt_id_bm を待ち、そのイベントが UTIL_SEQ_SetEvt() でセットされるまで戻らないようにシーケンサにリクエストします。
`void UTIL_SEQ_SetEvt( UTIL_SEQ_bm_t evt_id_bm )`	イベント evt_id_bm が発生したことをシーケンサに通知します（イベントは先にリクエストされている必要があります）。
`void UTIL_SEQ_EvtIdle(UTIL_SEQ_bm_t task_id_bm, UTIL_SEQ_bm_t evt_waited_bm)`	シーケンサが特定イベントを待っている間にコールします。
`void UTIL_SEQ_ClrEvt( UTIL_SEQ_bm_t evt_id_bm )`	保留されているイベントをクリアします。
`UTIL_SEQ_bm_t UTIL_SEQ_IsEvtPend( void )`	保留されているイベントの evt_id_bm を返します。

4.4.3 インタフェースと動作の詳細

シーケンサは、リクエストを受けたときに関数をコールする while ループのパッケージとなっています。

```
while(1)
{
   if(task_id1)
   {
      task_id1 = 0
      Fct1();
   }

   if (task_id2)
   {
      task_id2= 0;
      Fct2();
   }

   __disable_irq();
   If (!(task_id1|| task_id2))
   {
      UTIL_SEQ_Idle();
   }
   __enable_irq();
}
```

```
void UTIL_SEQ_Run(UTIL_SEQ_bm_t mask_bm)
```

while (1) ループの本体を実行します。mask_bm パラメータは、シーケンサが実行可能な関数のリストです。それぞれの関数は、その mask_bm の中の 1ビットに関連付けられています。起動の最後に、while (1) ループの中で mask_bm = (~0) としてこの API をコールして、保留されている関数のシーケンサによる実行を可能とする必要があります。

```
void UTIL_SEQ_Idle( void )
```

シーケンサに実行すべき関数がない場合に、クリティカル·セクション（CortexM PRIMASK ビットをセット - すべての割込みをマスク）の中でコールします。この場合、アプリケーションは低消費電力モードに移行する必要があります。

```
void UTIL_SEQ_RegTask(UTIL_SEQ_bm_t task_id_bm, uint32_t flags, void
(*task)( void ))
```

フラグ task_id_bm に関連付けられている関数タスクを while ループに追加するようにシーケンサに通知します。

```
void UTIL_SEQ_SetTask( UTIL_SEQ_bm_t task_id_bm , UTIL_SEQ_bm_t task_prio )
```

スケジューラにフラグ task_id_bm をセットして、関連付けられている関数をコールします。

task_prio は、次にどの関数をコールするか決定する必要がある場合にシーケンサによって評価されます。現在の関数の実行が完了した場合のみ行うことができます。複数の関数がフラグをセットしている場合、優先順位が最も高いものが実行されます（0が最高）。この API は、関数が実際に実行される前に異なる優先順位で複数回コールしても構いません。その場合、シーケンサは最も高い優先順位を記録します。関数が実行される前に API が何回コールされたとしても、シーケンサは関連付けられている関数を 1回限り実行します。

```
void UTIL_SEQ_PauseTask( UTIL_SEQ_bm_t task_id_bm ) :
```

たとえセットされても、フラグ task_id_bm に関連付けられている関数を実行しないようにシーケンサに通知します。UTIL_SEQ_PauseTask() の後に API UTIL_SEQ_SetTask() がコールされた場合、リクエストは記録されますが、関数は実行されません。UTIL_SEQ_PauseTask() に関連付けられているマスクは、void UTIL_SEQ_Run(UTIL_SEQ_bm_t mask_bm) に関連付けられているマスクとは無関係です。

関数が実行可能であるのは、そのフラグがセットされていて、両方のマスクで有効となっている（デフォルト）場合のみです。

```
void UTIL_SEQ_ResumeTask( UTIL_SEQ_bm_t task_id_bm ) :
```

UTIL_SEQ_PauseTask() からのリクエストをキャンセルします。UTIL_SEQ_PauseTask() がリクエストされていないときにこの API がコールされても影響はありません。

```
void UTIL_SEQ_WaitEvt( UTIL_SEQ_bm_t evt_id_bm )
```

この API がコールされると、関連付けられている evt_id_bm シグナルがセットされるまで戻りません。evt_id_bm の 32ビット値の中の 1ビットのみをセットする必要があります。シーケンサがこのイベントを待っている間に、イベント evt_id_bm に対する while ループの中で UTIL_SEQ_EvtIdle() をコールします。先に進む前にフラグに対するポーリングが行われている場合には、この関数を使用してすべてのコードを置き換える必要があります。

```
void UTIL_SEQ_SetEvt( UTIL_SEQ_bm_t evt_id_bm )
```

すでに UTIL_SEQ_WaitEvt() がコールされている場合のみコールする必要があります。関数 UTIL_SEQ_WaitEvt() が待っているシグナル evt_id_bm がセットされます。UTIL_SEQ_WaitEvt() 関数の前にこの API をコールすると、すでにフラグがセットされているため UTIL_SEQ_WaitEvt() に対するコールは直ちに戻ります。

```
void UTIL_SEQ_EvtIdle(UTIL_SEQ_bm_t task_id_bm, UTIL_SEQ_bm_t
evt_waited_bm)
```

UTIL_SEQ_SetEvt() でセットされるシグナルを API void UTIL_SEQ_WaitEvt() が待っている間にコールされます。

この API はシーケンサの中で弱く実行されて UTIL_SEQ_Run(0) をコールします。つまり、このイベントの発生を待っている間に、関数 UTIL_SEQ_Idle() によって、システムがフラグを待っている間に低消費電力モードに移行可能であることを意味します。

アプリケーションは、この API を実行して UTIL_SEQ_Run(mask_bm) に 0以外のパラメータを渡すことができます。mask_bm の中で 1にセットされた各ビットは、UTIL_SEQ_SetTask() でセットされた場合に、このフラグに関連付けられている関数を実行するようにシーケンサにリクエストします。つまり、関数 UTIL_SEQ_WaitEvt() がコールされてリクエストされたイベントから戻るのを待っている間に、フラグがセットされた場合にマスクされていない関数を実行することも、シーケンサによって実行が保留されているタスクがない場合に UTIL_SEQ_Idle() をコールすることもできます。

```
void UTIL_SEQ_ClrEvt( UTIL_SEQ_bm_t   evt_id_bm )
```

一部のアプリケーションでは、この API は、すでに Evt がセットされている間に API UTIL_SEQ_WaitEvt() がコールされる必要がある場合にコールすることもできます。その場合、Evt はクリアする必要があります。

```
UTIL_SEQ_bm_t UTIL_SEQ_IsEvtPend( void ):
```

この API は、現在保留されている Evt を返します。複数の UTIL_SEQ_WaitEvt() が入れ子になっている場合には、最後のものが返されます。つまり、最も深い UTIL_SEQ_WaitEvt() を行ったものがその呼び出し側に戻ります。

4.5 タイマ･サーバ

タイマ･サーバは、次のような機能を備えています。

- 使用可能な RAM 容量に応じて最大 255個の仮想タイマ
- シングルショットモードと繰り返しモード
- 仮想タイマを停止し、異なるタイムアウト値で再開
- タイマの削除
- タイムアウトは 1から 2^{32} - 1ティック

タイマ･サーバは、RTC ウェイクアップタイマを共有する複数の仮想タイマを提供します。それぞれの仮想タイマは、シングルショットタイマとしても繰り返しタイマとしても定義可能です。繰り返しタイマがサイクルの終わりに達すると、通知が行われて同じタイムアウトで仮想タイマが自動的に再開されます。シングルショットタイマが終了すると、通知が行われて仮想タイマは保留状態（登録は維持されておりいつでも再開可能）にセットされます。仮想タイマを停止して異なるタイムアウト値で再開することもできます。仮想タイマが必要ではなくなった場合には、削除してタイマ･サーバのスロットを開放する必要があります。

タイマ･サーバはカレンダと同時に使用できます。

4.5.1 実装

タイマ･サーバを使用するには、アプリケーションは以下の手順を実行する必要があります。

- RTC IP を設定します。アプリケーションにカレンダが必要である場合、RTC 設定はカレンダ設定要件に適合している必要があります。カレンダを使用しない場合、RTC はタイマ･サーバ専用に最適化することができます。
- HW_TS_Init() でタイマ･サーバを初期化します。
- HW_TS_RTC_Int_AppNot() を実行します（オプション）。これを実行しないと、タイマコールバックは、RTC 割込みハンドラ･コンテキストの中でコールされます。
- HW_TS_Create() で仮想タイマを作成します。
- HW_TS_Stop() と HW_TS_Start() で仮想タイマを使用します。
- 不要な場合には、HW_TS_Delete() を使用して仮想タイマを削除します。

4.5.2 インタフェース

表 4. インタフェース関数

機能	説明

`void HW_TS_Init(`
`HW_TS_InitMode_t TimerInitMode,`
`RTC_HandleTypeDef *hrtc);`
	タイマ･サーバを初期化します。

`HW_TS_ReturnStatus_t HW_TS_Create(`
`uint32_t TimerProcessID,`
`uint8_t *pTimerId,`
`HW_TS_Mode_t TimerMode,`
`HW_TS_pTimerCb_t pTimerCallBack)`
	仮想タイマを作成します。

`void HW_TS_Stop(uint8_t TimerID)`	仮想タイマを停止します。

`void HW_TS_Start(`
`uint8_t TimerID,`
`uint32_t timeout_ticks)`
	仮想タイマを開始します。

`void HW_TS_Delete(uint8_t TimerID`	仮想タイマを削除します。
`void HW_TS_RTC_Wakeup_Handler(void);`	タイマ･サーバ･ハンドラが RTC 割込みハンドラからコールされます。
`uint16_t` `HW_TS_RTC_ReadLeftTicksToCount(void);`	次の割込みまでにカウントするティック数を返します。

`void HW_TS_RTC_Int_AppNot(`
`uint32_t TimerProcessID,`
`uint8_t TimerID,`
`HW_TS_pTimerCb_t pTimerCallBack)`
	仮想タイマが終了したことをアプリケーションに報告します。

`void HW_TS_RTC_CountUpdated_AppNot(void)`	次の割込みまでのティック数がタイマ･サーバによって更新されたことをアプリケーションに報告します。

4.5.3 インタフェースと動作の詳細

タイマ･サーバは、システムが RUN から STANDBY モードまで動作する仮想タイマを提供します。

`void HW_TS_Init(HW_TS_InitMode_t TimerInitMode, RTC_HandleTypeDef *hrtc) :`

このコマンドは、あらかじめ行われている必要のある RTC IP 設定に基づいてタイマ･サーバを初期化します。

TimerInitMode は、タイマ･サーバのブートモードを選択します。STANDBY モードがサポートされていてデバイスが STANDBY からウェイクアップする場合には、タイマ･サーバ･コンテキストがリセットされないように、TimerInitMode を hw_ts_InitMode_Limited にセットします。そうでなければ、TimerInitMode を hw_ts_InitMode_Full にセットして完全な初期化を行う必要があります。

hrtc は Cube HAL RTCのハンドル型です。

```
HW_TS_ReturnStatus_t HW_TS_Create(uint32_t TimerProcessID,
uint8_t *pTimerId,
HW_TS_Mode_t TimerMode,
HW_TS_pTimerCb_t pTimerCallBack) :

pTimerId
```

これは、生成したタイマの停止／開始／削除に使用する必要のある、タイマ･サーバによって呼び出し側に戻される ID です。

`TimerMode`

タイマモードは、シングルショットモードと繰り返しモードのいずれかとなります。シングルショットモードでは、タイムアウトに達するとタイマは停止します。繰り返しモードでは、タイムアウトのたびに以前にプログラムされたものと同じ値で再開されます。このモードは、タイマ生成時に固定されます。モードを変更するには、タイマを削除して新しいタイマを生成する必要があります。この場合、新しく割り当てられた pTimerId は別の値である可能性があることに注意してください。

`pTimerCallBack`

タイムアウト時のユーザコールバックです。

`TimerProcessID`

これはユーザによって定義され、HW_TS_RTC_Int_AppNot() で使用されるためのものです。タイマが生成されたときに、割り当てられた ID を知っているのは呼び出し側のみです。TimerProcessID は、HW_TS_RTC_Int_AppNot() の実行時に適切な判断が行えるように、pTimerCallBack で HW_TS_RTC_Int_AppNot() の中で返されます。

`void HW_TS_Stop(uint8_t TimerID)`

タイマ TimerID を停止します。タイマが実行中ではない場合には影響はありません。タイマ TimerID は生成済みである必要があります。タイマが停止すると、（同一の TimerMode と同一の pTimerCallBack を持つ）同じタイマが異なる値で再開できるように、TimerID はタイマ･サーバに割り当てられたままとなります。

`void HW_TS_Start(uint8_t TimerID, uint32_t timeout_ticks)`

タイマ TimerID を timeout_ticks 値で開始します。timeout_ticks の値は RTC IP の設定に依存します。すでに TimerID が実行中である場合、最初にタイマ･サーバの中で停止された後に新しい timeout_ticks 値で再開されます。

`void HW_TS_Delete(uint8_t TimerID)`

タイマ･サーバから TimerID を削除します。その TimerID は、新しい仮想タイマに割り当てることができます。この API は、動作中の TimerID でコールすることもできます。その場合、最初に停止された後に削除されます。

`void HW_TS_RTC_Wakeup_Handler(void)`

この割込みハンドラは、RTC 割込みハンドラでアプリケーションによってコールされる必要があります。このハンドラは、RTC と EXTI ペリフェラルのすべての必要なステータスフラグをクリアします。

`uint16_t HW_TS_RTC_ReadLeftTicksToCount(void)`

この API は、割込みがタイマ･サーバによって生成される前にカウントされる残りティック数を返します。これは、システムが低消費電力モードに移行する際に、次のウェイクアップがいつ予定されているか次第で、どの低消費電力モードを適用するか決定する必要がある場合に使用することができます。

タイマが無効となっている（リストにタイマがない）場合、0xFFFF が返されます。

```
void HW_TS_RTC_Int_AppNot(uint32_t TimerProcessID,
uint8_t TimerID,
HW_TS_pTimerCb_t pTimerCallBack)
```

この API は、ユーザアプリケーションによって実行される必要があります。

タイマが終了したときにアプリケーションに通知します。この API は RTC ウェイクアップ割込みコンテキストの中で動作しており、どれだけのコードが pTimerCallBack の中で実行されるか次第では、アプリケーションは、バックグラウンドタスクとして pTimerCallBack をコールする方を好むことがあります。TimerID を知っているのは呼び出し側のみである限り、TimerProcessID を使用してこの pTimerCallBack がどのモジュールに属しているかを特定することが可能であり、アプリケーションは、RTC ウェイクアップ割込みコンテキストの中でそれがコール可能かどうかを評価することができます。

```
void HW_TS_RTC_CountUpdated_AppNot(void):
```

この API は、ユーザアプリケーションによって実行される必要があります。

この API は、カウンタが更新されていることをアプリケーションに通知します。
HW_TS_RTC_ReadLeftTicksToCount () API とともに使用するためのものです。カウンタは、最後に HW_TS_RTC_ReadLeftTicksToCount () をコールしてから、低消費電力モードに移行する前に更新されている可能性があります。この通知は、低消費電力モードに移行する前に競合状態を解決して、カウンタ値を再評価する方法をアプリケーションに提供します。

4.6 低消費電力マネージャ

低消費電力マネージャは、最大 32のさまざまなユーザから入力を受け取って、システムが使用可能な最も低い電源モードを計算するシンプルなインタフェースを提供します。また、低消費電力モードに移行する前またはそれを終了するときのフックをアプリケーションに提供します。

低消費電力マネージャは、次のような機能を備えています。

- 最大 32ユーザまで
- STOP モードと OFF モード（STANDBY と SHUTDOWN）
- 低消費電力モード選択
- 低消費電力モードの実行
- 低消費電力モードへの移行時と終了時のコールバック
- RUN モードはサポートされていません。アプリケーションがこのモードを継続する必要がある場合には、UTIL_LPM_EnterModeSelected () をコールしないでください。

4.6.1 実装

低消費電力マネージャは、さまざまな低消費電力モードリクエストを持つ最大 32ユーザまで処理可能です。

低消費電力マネージャを使用するには、アプリケーションは以下の手順を実行する必要があります。

- ユーザ ID を生成します。
- リクエストされた低消費電力モードをセットするには、定義されたユーザ ID で UTIL_LPM_SetOffMode() と UTIL_LPM_SetStopMode() のいずれかをいつでもコールします。
- バックグラウンドで void UTIL_LPM_EnterLowPower() をコールします。

4.6.2 インタフェース

表 5. インタフェース関数

機能	説明
`UTIL_LPM_ModeSelected_t UTIL_LPM_ReadModeSel(void)`	適用される選択された低消費電力モードを返します。
`UTIL_LPM_SetOffMode(UTIL_LPM_bm_t lpm_id_bm, UTIL_LPM_State_t state)`	どのユーザに対しても随時 OFF モードを有効または無効にします。
`void UTIL_LPM_SetStopMode(UTIL_LPM_bm_t lpm_id_bm, UTIL_LPM_State_t state)`	どのユーザに対しても随時 STOP モードを有効または無効にします。
`void UTIL_LPM_EnterLowPower(void)`	選択されている省電力モードに移行します。
`void UTIL_LPM_EnterSleepMode(void)`	SLEEP モードに移行する前に API をコールします。
`void UTIL_LPM_ExitSleepMode(void)`	SLEEP モードの終了時に API をコールします。
`void UTIL_LPM_EnterStopMode(void)`	STOP モードに移行する前に API をコールします。
`void UTIL_LPM_ExitStopMode(void);`	STOP モードの終了時に API をコールします。
`void UTIL_LPM_EnterOffMode(void)`	OFF モードに移行する前に API をコールします。
`void UTIL_LPM_ExitOffMode(void);`	OFF モードの終了時に API をコールします。MCU が 期待どおりに OFF モードに移行しなかった場合にのみコールします。

4.7 Flash メモリ管理

STM32WB は CPU1 と CPU2 との間で 1つのシングルバンクを共有します。Flash メモリが書き込まれたり消去されたりするときに、そこから命令をフェッチする方法はありません。

CPU が Flash メモリからコードを実行した場合、書込みまたは消去の操作が始まると直ちにストールします。

CPU が SRAM からコードを実行した場合、CPU は書込みまたは消去の操作の進行中にストールしません（Flash メモリからデータを読み出さないものと仮定します）。

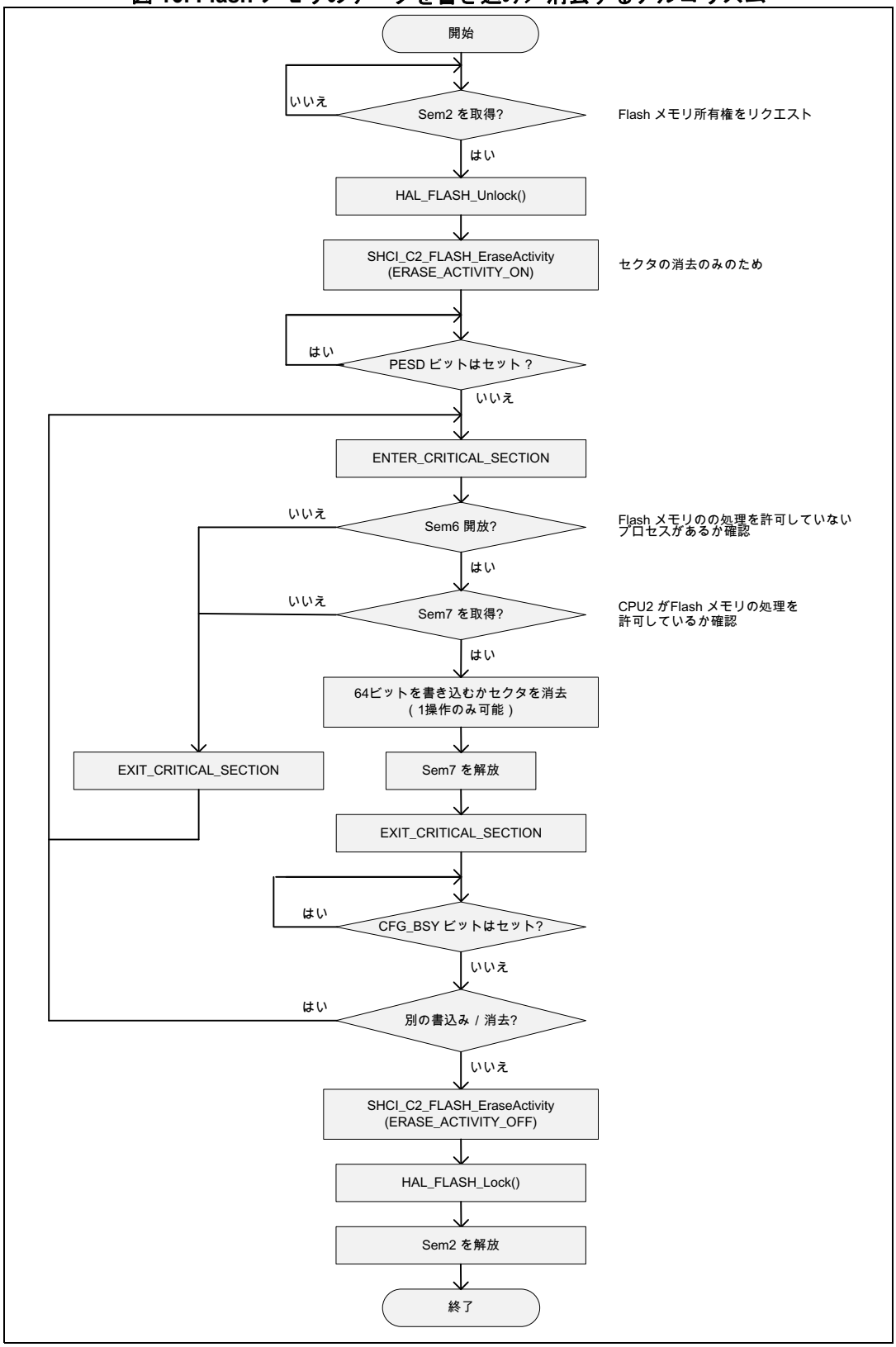
4.7.1 CPU2 タイミング保護

セキュリティ上の理由により、CPU2 は SRAM からいかなるコードも実行できないようになっています。CPU2 のタイミングを保護するため、Sem7 を使用して CPU1 からリクエストされた Flash メモリ操作を有効化または無効化します。

CPU1 上のアプリケーションは、図 10に示したアルゴリズムを実行して Flash メモリの書込みまたは消去を行う必要があります。また、（図 10に定義されているクリティカル·セクションの外で）次の動作を自分自身のドライバにて実行する必要があります。

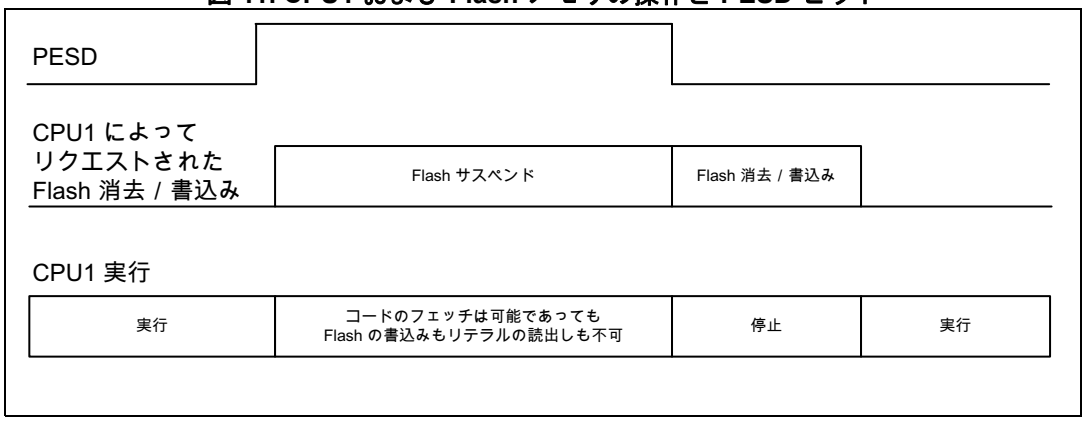
- Flash メモリに何らかアクセスする前に Sem2 を取得し、IP へのアクセスに不要となったら解放します。
- ユーザドライバがセクタを消去する必要がある場合には、まずコマンド SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_ON) を送信する必要があります。関係するセクタがすべて消去されたら、コマンド SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_OFF) を送信する必要があります（セクション 4.7.1 : 参照）。
- CPU2 タイミング保護に（デフォルトの）PESD ビットメカニズムが使用されている場合（セクション 4.7.1 : 参照）、FLASH ドライバは、FLASH_SR レジスタから CFGBSY ビットをポーリングするか、値が書き込まれるべきものになるまでメモリを読み直すかする必要があります。

図 10. Flash メモリのデータを書き込み／消去するアルゴリズム



デフォルトで CPU2 は、Sem7 ではなく、(FLASH_SR レジスタからの) PESD ビットメカニズムを使用してその Bluetooth LE タイミングを保護します。Sem7 のチェックは不要ですが、アルゴリズムは依然として有効です。その欠点は、CPU1 が書込みまたは消去の操作を開始するのと同時に CPU2 によって PESD ビットがセットされた場合に、CPU1 はコードのフェッチは可能ですが、実行されるコードにたとえ必要であったとしても、メモリから正しい値を読み出せないことです。PESD メカニズムが使用された場合に CPU1 がストールするかどうかを制御するのは非常に困難です。さらには、CPU2 による PESD ビットの解放に対する割込み信号は存在しませんので、非同期のソフトウェアフローは不可能となります。

図 11. CPU1 および Flash メモリの操作と PESD ビット



Bluetooth LE タイミングを保護するための CPU2 による PESD または Sem7 メカニズムの使用は、システムコマンド SHCI_C2_SetFlashActivityControl() を用いて CPU1 から設定可能です。このコマンドはいつでも送信可能ですが、初期化フェーズ中に送信することを推奨します。

デフォルトで、CPU2 は、CPU1 からリクエストされた書込み操作に対するタイミングを保護します。CPU1 が消去操作を開始する必要がある場合には、まずシステムコマンド SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_ON) を送信する必要があります。それ以上消去操作をリクエストすることが予想されない場合、システムコマンド SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_OFF) を送信する必要があります。これらのコマンドは、1回の消去操作のたびに送信する必要はありません。最初の消去操作をリクエストする前に保護を有効化して、最後の消去操作が実行された後に保護を無効化することを推奨します。

4.7.2 CPU1 タイミング保護

CPU2 からリクエストされた Flash メモリ操作（書込みまたは消去）のためにストールしないことを CPU1が確認する必要がある場合、Sem6 を取得する必要があります。Sem6 が解放されるまでは、CPU2 はいかなる Flash メモリ操作もリクエストしません。

Sem6 が取得されたときには、Flash メモリ操作を実行する処理に CPU 2がすでに入っていることを意味します（処理の開始直前または終了直後）。CPU2 からいかなる Flash メモリ操作もリクエストされないようにする必要がある場合、CPU1 は Sem6 をポーリングして取得する必要があります。

CPU2 は、図 10 に記載されているものと同じアルゴリズムを使用します。

4.7.3 RF 動作と Flash メモリ管理の競合

CPU2 が Flash メモリ操作を開始しないようにするために CPU1 が Sem6 を使用するのではない場合ですら、CPU2 が消去操作を開始できないユースケースがいくつか存在します。

Flash メモリ書込み操作はいつでも可能ですが、CPU2 上で NVM がフルである場合、Flash メモリ書込みリクエストには消去操作が必要となる可能性があります。この場合、消去操作が実行されるまでデータは書き込まれません。

コマンド SHCI_C2_FLASH_EraseActivity(ERASE_ACTIVITY_ON) によって CPU1 から通知を受けているか、内部的に Flash メモリ消去操作を実行して NVM のセクタをいくつか消去する必要があるという理由により、CPU2 が消去からそのタイミングを保護する必要がある場合には、無線動作の前 25ms からその終了まではいかなるメモリ操作も禁止されます。Bluetooth LE の動作中に Flash メモリ消去操作を実行するには、アプリケーションは、25ms を上回る無線アイドル時間が存在することを確認する必要があります。

- Bluetooth LE アドバタイジング：Flash メモリ消去操作を実行可能であるために、アドバタイジング間隔は 25ms + アドバタイジング･パケット長よりも長い必要があります。
- Bluetooth LE 接続済み：Flash メモリ消去操作を実行可能であるために、接続間隔は 25ms + パケット長よりも長い必要があります。
- データ･スループット･ユースケース：データ･ストリーミング･パケットの送信時には、2 回の接続間隔の間に可能な限り多くのデータを送信するために、無線はアクティブなままとなります。したがって、無線は消去操作に収まるほど十分に長くアイドル状態とはならない可能性があります。デバイスがマスタである場合、（aci_gap_create_connection() と aci_gap_start_connection_update() のいずれかのコマンドで）Connection Event Length パラメータを小さくして、2回の接続間隔の間にデバイスが完全に一杯となることを防止することができます。デバイスがスレーブである場合、送信するデータが 2回の接続イベントの間隔の一部のみに収まるように接続間隔を延ばすことを、マスタにリクエストする必要があります。

CPU2 が Flash メモリにデータを書き込む必要のあるのは以下の場合に限られます。

- ペアリングフェーズの後、セキュリティ情報を格納するため
- 切断後、切断されたばかりのクライアントの GATT クライアント･ディスクリプタ情報を格納するため（ボンディングされていた場合）

4.8 CPU からのデバッグ情報

4.8.1 GPIO

バックグラウンドタスク、割込みハンドラ、Bluetooth LE IP Core 信号などの CPU2 のリアルタイム動作の多くは GPIO に出力できます。HW によって駆動される Bluetooth LE IP Core 信号を除くと、ある信号の特定 GPIO への割当ては CPU1 側から完全に設定可能となっています。したがって、Bluetooth LE IP Core GPIO は、アプリケーションによって使用されない場合のみ有効化される必要があります。全体の設定は、\Core\Src にあるファイル app_debug.c の中でアプリケーションごとに行います。

HW 信号

aRfConfigList[] テーブルには、無線動作に応じてハードウェアにより駆動される GPIO のリストが記載されています。それぞれの信号に対して監視すべき 4個のパラメータがあります。

```
{ GPIOA, LL_GPIO_PIN_9, 0, 0},      /* DTB13 - Tx/Rx Start */
```

最初の 2つのパラメータは、使用する GPIO を定義します（この例では、PA9 は出力 DTB13 に使用されます）。これらの 2つのパラメータは変更できません。信号を監視するには、関連付けられている GPIO がボード上にある必要があります。

3番目のパラメータは、信号を有効化（1）または無効化（0）するために使用します。すべての信号は、デフォルトで 0にセットされます。

4番目のパラメータは未使用であり、0に保持するものとします。

関連付けられている GPIO の信号を監視するには、3番目のパラメータを 1にセットして、ファイルの先頭の BLE_DTB_CFG コンパイラ·スイッチを 7にセットする必要があります。

最も有用な信号は DTB13 であり、すべての無線動作が表されます。

SW 信号

aGpioConfigList [] テーブルには、ソフトウェアにより駆動される GPIO のリストが記載されています。それぞれの信号に対して監視すべき 4個のパラメータがあります。

```
{ GPIOA, LL_GPIO_PIN_0, 0, 0},  /* BLE_ISR - Set on Entry / Reset on Exit */
```

最初の 2つのパラメータは、信号の出力に使用する GPIO を定義します。これらは完全に設定可能です。どの GPIO も、アプリケーションの中で未使用を選択できます。

3番目のパラメータは、信号を有効化（1）または無効化（0）するために使用します。すべての信号は、デフォルトで 0にセットされます。

4番目のパラメータは未使用であり、0に保持する必要があります。

関連付けられている GPIO の信号を監視するには、3番目のパラメータを 1にセットする必要があります。

4.8.2 SRAM2

ハードフォールト

CPU2 がハードフォールト割込みハンドラに移行したときには、無限ループを実行する前にさまざまな情報を出力できます。

app_debug.c - aGpioConfigList [] の中で有効にされている場合、GPIO のセットが可能です。

以下のデータを SRAM2A に書き込みます。

@SRAM2A_BASE：0x1170FD0F	ハードフォールト問題を特定するキーワード
@SRAM2A_BASE + 4	ハードフォールトを生成したプログラムカウンタ値
@SRAM2A_BASE + 8	ハードフォールトを生成した命令が実行されたときのリンク·レジスタ値
@SRAM2A_BASE + 12	ハードフォールトを生成した命令が実行されたときのスタック·ポインタ値

セキュリティ攻撃

メールボックス経由でデータを交換するために CPU2 が備えているバッファが非セキュアな SRAM2 の中ではない場合、CPU2 は無限ループに移行し、キーワード 0x3DE96F61を SRAM2A_BASE に書き込みます。

4.9 FreeRTOS 低消費電力

CPU2 上で動作するスタックが何であれ、FreeRTOS 低消費電力モードでは、すべてのワイヤレス・アプリケーションに対して CPU1 上の同じ実装が共有されます。

FreeRTOS に予約済みの SysTick と衝突しないように、HAL ティックは TIM17 にマッピングされます。\Applications\BLE\BLE_HeartRateFreeRTOS\Core\Src のファイル stm32wbxx_hal_timebase_tim.c には、以下の HAL 関数が実装されています。

- HAL_InitTick()
- HAL_SuspendTick()
- HAL_ResumeTick()

HAL によって使用されるティックをインクリメントするために、TIM17 ユーザ割込みハンドラ HAL_TIM_PeriodElapsedCallback() が main.c の中に実装されています。この実装をカスタマイズして、別のタイマを選択することもできます。

FreeRTOS がアイドルモードにある場合、SysTick はオフとなり、低消費電力タイマと置き換えられます。\Applications\BLE\BLE_HeartRateFreeRTOS\Core\Src のファイル freertos_port.c には、ティックレスモードが実装されています。

- STM32WB デバイスで使用可能である低消費電力モードをベースとするティックレスモードをサポートするために、vPortSuppressTicksAndSleep() が再実行されます。
- STM32WB デバイスで使用可能である低消費電力タイマを起動するために、vPortSetupTimerInterrupt() が再実行されます。

現在の実装では、RTC 上で動作するタイマ・サーバが使用されています。タイマの選択は、以下の関数を再実行することにより変更できます。

- LpTimerInit() : 使用する低消費電力モードを初期化します。
- LpTimerCb() : 単なるウェイクアップ以上のものが必要な場合に使用します。現在の実装では、ウェイクアップ時に行われるすべての動作は、vPortSuppressTicksAndSleep() の中の低消費電力モードの終了時に実行されており、タイマ・コールバックの中ではありません。
- LpTimerStart() : 低消費電力モードに移行する前に低消費電力モードを起動します。
- LpGetElapsedTime() : システムが低消費電力モードにある時間を返します。FreeRTOS によって使用されている SysTick を vTaskStepTick() で更新するために必要となります。

低消費電力モードには LpEnter() を用いて移行します。現在の実装は、FreeRTOS に基づいているかどうかによらず、すべての Bluetooth LE アプリケーションで使用されている低消費電力マネージャをベースとしています。LpEnter() の実装はカスタマイズ可能です。

Bluetooth LE

バックグラウンドでコールされる関数の数はアプリケーションに依存しますが、それぞれの関数が専用タスクによってコールされるか、ある 1つの共通タスクからコールされるかについてもアプリケーションによって決まります。Bluetooth LE アーキテクチャは、どのような組み合わせもサポートしています。

Bluetooth LEアプリケーションが何であれ、あるタスクの中には少なくとも次の 2個の関数が存在する必要があります。

- **hci_user_evt_proc()** : ミドルウェアから hci_notify_asynch_evt() がコールされる場合、バックグラウンドでこの関数がコールされる必要があります。hci_user_evt_proc() は、IPCC 割込みコンテキストからコールされる可能性がある hci_notify_asynch_evt() の中からコールしないでください。ミドルウェアから hci_notify_asynch_evt() がコールされる時間と、バックグラウンドで hci_user_evt_proc() がコールされる時間の間には、タイミングの制約はありません。ただし、一部のデータ・スループット・ユースケースでは、通知を受けた同一速度でイベントの読出しが

行えるほど十分に短い時間である場合に、パフォーマンスが向上します。複数の hci_notify_asynch_evt() を受信した場合、hci_user_evt_proc() 関数をバックグラウンドでコールする必要があるのは 1回のみです。バックグラウンドから複数回 hci_user_evt_proc() をコールしても無害ですが、hci_notify_asynch_evt() による通知は 1回のみであるか、全くありません。

- **shci_user_evt_proc()**：要件は、hci_user_evt_proc() およびそれに関連付けられている通知 shci_notify_asynch_evt() と同一です。現在このシステムチャネルにおけるデータ·スループットは存在しません。

すでに保留中のものが存在する間は Bluetooth LE コマンドの送信が行えないか、すでに保留中のものが存在する間はシステムコマンドの送信が行えない限り、アプリケーションがセマフォ·メカニズムを実装できるように、ミドルウェアがフックを提供します。

ミドルウェアから hci_cmd_resp_wait() がコールされたときにセマフォを取得して、hci_cmd_resp_release() の受信時に解放する必要があります。セマフォが解放されるまで、アプリケーションは hci_cmd_resp_wait() から戻ってはなりません。

システムチャネル上で shci_cmd_resp_wait()/shci_cmd_resp_release() を用いて同一メカニズムを処理するには、別のセマフォを使用する必要があります。

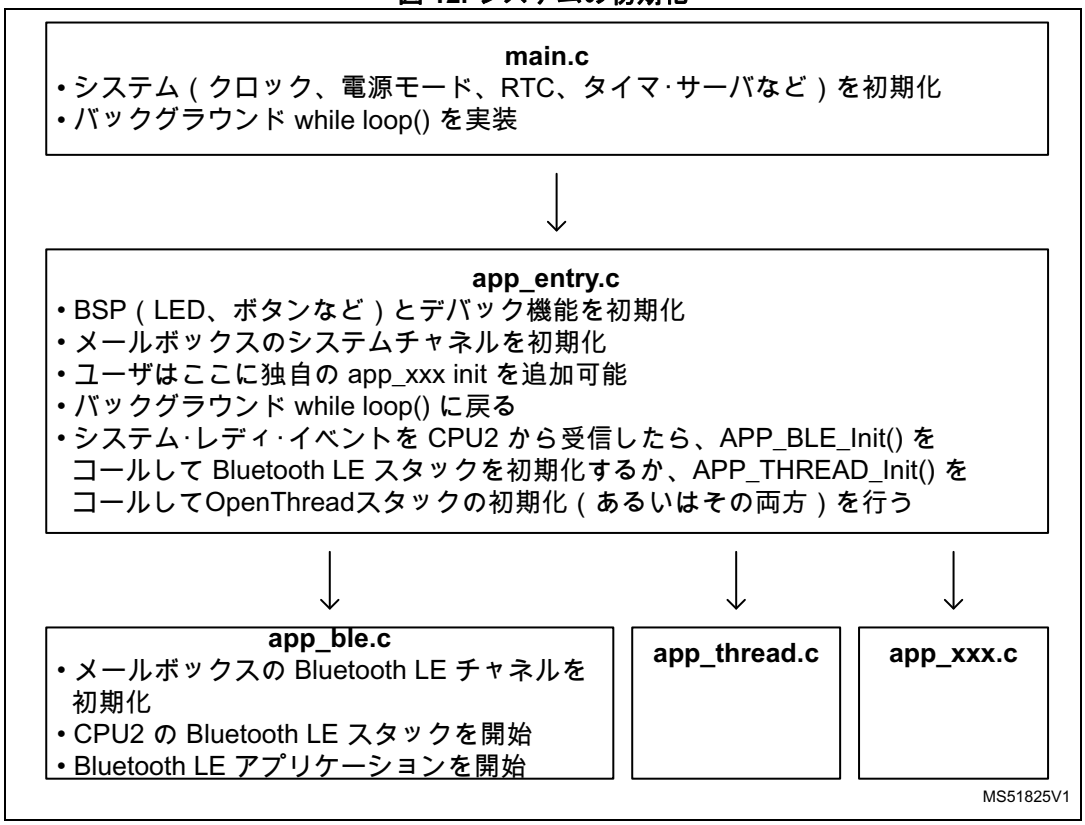
参考資料

5 システムの初期化

すべてのアプリケーションは、3セットのファイルから始まります（図 12 参照）。

1. main.c : あらゆるアプリケーションに共通のすべての HW 設定（CPU2に供給されるクロックは必ず 32MHz である必要があります）
2. app_entry.c : あらゆるアプリケーションに共通のすべての SW 設定と実装
3. app_ble.c / app_thread.c / app_xxx.c : アプリケーション専用ファイル

図 12. システムの初期化



参考資料

6 Bluetooth LE アプリケーションのステップバイステップ設計

このセクションでは、STM32WB デバイスに Bluetooth LE アプリケーションを設計して実装する方法に関する情報とコード例について説明します。

6.1 初期化フェーズ

アプリケーションの初期化には、いくつかの手順が必ず必要となります。

- デバイス（HAL、リセット・デバイス、クロックと電源設定）を初期化。
- プラットフォーム（ボタン、LED）を設定
- ハードウェア（UART、デバッグ）を設定
- Bluetooth LE デバイスのパブリック・アドレス（使用する場合）を設定
  - `aci_hal_write_config_data()` API
- Tx 電源を設定
  - `aci_hal_set_tx_power_level()` API
- Bluetooth LE GATT レイヤを初期化
  - `aci_gatt_init()` API
- 選択したデバイスの役割に応じて Bluetooth LE GAP レイヤを初期化
  - `aci_gap_init("role")API`
- 適切なセキュリティ I/O 機能と認証要件を設定（Bluetooth LE セキュリティを使用する場合）
  - `aci_gap_set_io_capability()` および `aci_gap_set_authentication_requirement()` API
- デバイスが GATT サーバである場合には必要なサービス、キャラクタリスティック、キャラクタリスティック・ディスクリプタを定義
  - `aci_gatt_add_service()`、`aci_gatt_add_char()`、`aci_gatt_add_char_desc()` API
- シーケンサを使用してタスクと低消費電力を管理

6.2 アドバタイジング・フェーズ（GAP ペリフェラル）

Bluetooth LE GAP セントラル（マスタ）デバイスと Bluetooth LE GAP ペリフェラル（スレーブ）デバイスの間の接続を確立するには、ペリフェラルデバイス上で GAP ディスカバリ可能モードを起動する必要があります。表 6 の API を使用できます。

表 6. アドバタイジング･フェーズ API の説明

API 名	説明
aci_gap_set_discoverable()	デバイスを一般ディスカバリ可能モードに設定します。 デバイスは aci_gap_set_non_discoverable() API を発行するまでディスカバリ可能です。
aci_gap_set_discoverable()	デバイスを限定ディスカバリ可能モードに設定します。デバイスは、最長期間 TGAP (lim_adv_timeout) である 180秒間ディスカバリ可能です。 アドバタイジングは、aci_gap_set_non_discoverable() API をコールしていつでも無効化できます。
aci_gap_set_direct_connectable()	デバイスを直接接続（ダイレクト・コネクション）モードに設定します。デバイスは、1.28秒だけ直接接続（ダイレクト・コネクション）モードにあります。その期間の中で接続が確立されなかった場合には、デバイスは非ディスカバリ可能モードに移行し、アドバタイジングは明示的に再度有効化される必要があります。
aci_gap_set_non_connectable()	デバイスを非接続可能(non-connectible)モードとします
aci_gap_set_undirect_connectable()	デバイスを不特定接続可能(undirected connectible)モードとします。

6.3 ディスカバリ・接続可能フェーズ（GAP セントラル）

2台のデバイス間の接続を生成するために、GAP セントラルはリモートをディスカバーした後に、対象デバイスへの接続を開始することが可能です。特定デバイスに対して直接接続を開始することも可能です。

GAP ディスカバリ手順に使用可能な API のリストを表 7 に示します。

表 7. GAP セントラル API

API	説明
aci_gap_start_limited_discovery_proc ()	リミテッド･ディスカバリ手順を開始します。コントローラはアクティブ・スキャンを開始します。リミテッド・リカバリ可能モードにあるデバイスのみが上位レイヤに戻されます。
aci_gap_start_general_discovery_proc ()	ジェネラル･ディスカバリ手順を開始します。コントローラはアクティブ・スキャンを開始します。
以下の API は、GAP 接続を確立する手順に使用できます。	
aci_gap_start_auto_connection_establish_proc ()	自動接続（オート・コネクション）手順を開始します。指定されたデバイスがコントローラ・ホワイトリストに追加され、「ホワイトリストを使用してどのアドバタイザに接続するか決定する」に設定されたイニシエータ・フィルタ・ポリシーを使用して GAP コントローラに対する接続コールを開始します。
aci_gap_create_connection ()	直接接続（ダイレクト・コネクション）手順を開始します。「ホワイトリストを無視して、指定デバイスのみに対する接続可能アドバタイジング・パケットを処理する」に設定されたイニシエータ・フィルタ・ポリシーを使用して、GAP による接続生成コールがコントローラに対して行われます。

表 7. GAP セントラル API（続き）

API	説明
aci_gap_start_auto_connection_establish_proc ()	自動接続（オート・コネクション）手順を開始します。指定されたデバイスがコントローラ・ホワイトリストに追加され、「ホワイトリストを使用してどのアドバタイザに接続するか決定する」に設定されたイニシエータ・フィルタ・ポリシーを使用して、GAP による接続生成コールがコントローラに対して行われます。
aci_gap_start_general_connection_establish_proc()	一般接続（ジェネラル・コネクション）手順を開始します。デバイスは、「すべてのアドバタイジング・パケットを受け入れる」に設定されたスキャナ・フィルタ・ポリシーを使用してコントローラのスキャンを有効化し、そのスキャン結果から、イベント・コールバック hci_le_advertising_report_event() を使用してすべてのデバイスが上位レイヤに送信されます。
aci_gap_start_selective_connection_establish_proc ()	選択的接続（セレクティブ・コネクション）手順を開始します。GAP は、指定されたデバイスのアドレスをホワイトリストに追加し、「ホワイトリストにあるデバイスからのパケットのみ受け入れる」に設定されたスキャナ・フィルタ・ポリシーを使用してコントローラのスキャンを有効化します。発見されたすべてのデバイスは、イベント・コールバック hci_le_advertising_report_event() によって上位レイヤに送信されます。
aci_gap_terminate_gap_proc()	指定された GAP 手順を終了します。

6.4 サービスとキャラクタリスティックの設定（GATT サーバ）

サービスおよびそれに関連するキャラクタリスティックを追加するために、ユーザアプリケーションはここに定義されている 2つのプロファイルから 1つを選択してください。

- Bluetooth SIG によって定義された標準プロファイル
  プロファイルの仕様およびサービス、ならびに、関連する定義済みのプロファイル、サービス、キャラクタリスティックと 16ビット UUID を使用してそれらを実装するためのキャラクタリスティック仕様書に従う必要があります（Bluetooth SIG Web ページ参照）。
- 独自の非標準プロファイル
  カスタムのサービスとキャラクタリスティックを定義する必要があります。この場合、128ビット UIDS が要求され、プロファイル実装者によって生成される必要があります（UUID 生成器 Web ページ www.famkruithof.net 参照）。

サービスは、次のプロシージャで追加できます。

```
aci_gatt_add_service(uint8_t Service_UUID_Type,
Service_UUID_t *Service_UUID,
uint8_t Service_Type,
uint8_t Max_Attribute_Records,
uint16_t *Service_Handle);
```

このプロシージャによってサービス・ハンドル（Service_Handle）へのポインタが返され、このポインタはユーザアプリケーション内部のサービスの特定に使用されます。キャラクタリスティックは、次のプロシージャでこのサービスに追加できます。

```
aci_gatt_add_char(uint16_t Service_Handle,
uint8_t Char_UUID_Type,
Char_UUID_t *Char_UUID,
```

```
uint8_t Char_Value_Length,
uint8_t Char_Properties,
uint8_t Security_Permissions,
uint8_t GATT_Evt_Mask,
uint8_t Enc_Key_Size,
uint8_t Is_Variable,
uint16_t *Char_Handle);
```

このプロシージャによってキャラクタリスティック·ハンドル（Char_Handle）へのポインタが返され、このポインタはユーザアプリケーション内部のキャラクタリスティックの特定に使用されます。

キャラクタリスティック所有者が Notify モードまたは Indicate モードで有効化されている場合、GATT サーバ側は次の API を使用して GATT クライアントに通知または通告を送信します。

`aci_gatt_update_char_value()`

6.5 サービスとキャラクタリスティックのディスカバリ（GATT クライアント）

2台のデバイスがひとたび接続されると、アプリケーションデータの交換は GATT クライアント·サーバ・アーキテクチャに基づくものとなります。

1台のデバイスは、F を実装して GATT クライアントを削除する必要があります。

GATT クライアントは、サービスとキャラクタリスティックのディスカバー、GATT サーバに対する通知／通告の有効化／無効化、キャラクタリスティックの書込みと読取り、および GATT サーバ通知の確認に次の API を使用します。

表 8. GATT クライアント API

API	説明
`aci_gatt_disc_all_primary_services ()`	GATT サーバ上のすべてのプライマリ・サービスをディスカバーする GATT クライアント手順を開始します。GATT クライアントがあるデバイスに接続し、そのデバイス上に提供されているすべてのプライマリ・サービスを発見して、それが何を行えるか判断したい場合に使用されます。 手順応答は、aci_att_read_by_group_type_resp_event() イベント・コールバックを通じて与えられます。
`aci_gatt_disc_primary_service_by_uuid()`	UUID を使用して GATT サーバ上のあるプライマリ・サービスをディスカバーする GATT クライアント手順を開始します。GATT クライアントがあるデバイスに接続し、他のあらゆるサービスを必要とせずにある特定のサービスを発見したい場合に使用されます。 手順応答は、aci_att_find_by_type_value_resp_event() イベント・コールバックを通じて与えられます。

表 8. GATT クライアント API（続き）

API	説明
aci_gatt_find_included_services()	含まれているすべてのサービスを発見する手順を開始します。GATT クライアントが、プライマリ・サービスがディスカバーされた後にセカンダリ・サービスをディスカバーしたい場合に使用されます。 手順応答は、aci_att_read_by_type_resp_event() イベント・コールバックを通じて与えられます。
aci_gatt_disc_all_char_of_service()	ある任意のサービスのすべてのキャラクタリスティックをディスカバーする GATT 手順を開始します。 手順応答は、aci_att_read_by_type_resp_event() イベント・コールバックを通じて与えられます。
aci_gatt_disc_char_by_uuid()	UUID によって指定されたすべてのキャラクタリスティックをディスカバーする GATT 手順を開始します。 手順応答は、aci_gatt_disc_read_char_by_uuid_resp_event() イベント・コールバックを通じて与えられます。
aci_gatt_disc_all_char_desc()	GATT サーバ上のすべてのキャラクタリスティック·ディスクリプタをディスカバーする手順を開始します。 応答は、aci_att_find_info_resp_event() イベント・コールバックを通じて与えられます。

すべてのコマンドにおいて、手順の最後は aci_gatt_proc_complete_event() イベント・コールバックによって示されています。

6.6 セキュリティ（ペアリングとボンディング）

Bluetooth LE セキュリティモデルには 5つのセキュリティ機能が含まれています。

1. ペアリング：1個または複数の共有秘密鍵を生成するプロセス
2. ボンディング：信頼できるデバイスペアを構成するために、ペアリング中に生成されたキーを後の接続で使用するために格納する行為
3. デバイス認証：2台のデバイスが同じキーを保有していることを確認するための検証
4. 暗号化：メッセージに機密性を提供
5. メッセージ整合性：メッセージの偽造に対する保護（4バイト・メッセージ整合性チェック（MIC））

Bluetooth LE では次の 4種類のペアリング方法が使用されます。

1. ジャストワークス
2. アウトオブバンド
3. パスキー入力
4. Bluetooth 4.2 以降の数値比較（セキュア接続のみ）

セキュリティキーの計算結果の判定方法

- レガシー暗号化 - ショートー時キー（STK）。STK は接続の暗号化のために生成されます。ボンディングの場合、その後の接続には LTK が使用されます。
- セキュア接続 - ロングタームキー（LTK）。LTK は接続の暗号化のために生成されます。

6.6.1 セキュリティモードとレベル

LE セキュリティモード 1（Link レイヤ):

- セキュリティなし - レベル 1
- 認証なし暗号化ペアリング - レベル 2
- 認証あり暗号化ペアリング - レベル 3
- 認証あり暗号化 LE セキュア接続ペアリング（BT 4.2）- レベル 4

認証ありペアリング：ペアリングは中間者攻撃（MITM）保護ありで行われます

認証なしペアリング：ペアリングは MITM 保護なしで行われます

LE セキュリティモード 2（ATT レイヤ) : サポートされません

- データ署名あり認証なしペアリング
- データ署名あり認証ありペアリング

6.6.2 セキュリティ・コマンド

デバイス初期化フェーズにおいて、次のコマンドでセキュリティ・プロパティを初期化できます。

`aci_gap_set_io_capability()`

デバイスの IO 機能をセットします。このコマンドは、デバイスが接続状態ではないときにのみ行う必要があります。

`aci_gap_set_authentication_requirement()`

デバイスに対する認証要件をセットします。このコマンドは、デバイスが接続状態ではないときにのみ行う必要があります。

このコマンドによって、ボンディングモード情報、MITM モード、LE セキュア接続サポート値、キー押し通知サポート値、暗号化キーサイズ、固定ピン使用の有無、ID アドレスタイプが定義されます。

- SC_Support パラメータによって、LE セキュア接続サポート値が定義されます。
  - 0x00：セキュア接続ペアリングはサポートされていません（レガシーペアリングモード)。
  - 0x01：セキュア接続ペアリングはサポートされていますがオプションです。
  - 0x02：セキュア接続ペアリングはサポートされていて必須です（SC のみモード)。

接続が確立された後に、次のようにセキュリティ手順を開始できます。

- マスタが aci_gap_set_pairing_req() を使用
  - SM ペアリング・リクエストを送信してペアリング処理を開始します。認証要件と IO 機能は、このコマンドの発行前にセットする必要があります。
  - force_rebond パラメータ値によって、デバイスがかつてボンディングされていてもペアリング・リクエストが送信されるかどうかが決まります。
- スレーブが aci_gap_slave_security_req() を使用
  - マスタにスレーブ・セキュリティ・リクエストを送信します。このコマンドを発行して、スレーブのセキュリティ要件をマスタに通知する必要があります。マスタは、リンクを暗号化することも、ペアリング手順を開始することも、リクエストを拒否することもできます。
  - ペアリング処理が完了すると aci_gap_pairing_complete_event が返されます。

デバイスは、SC_Support パラメータ値に応じて表 9 に記載されているコマンドのいずれかを使用して、セキュリティ・リクエストに対する回答を行います。

表 9. セキュリティ・コマンド

コマンド	説明
aci_gap_pass_key_resp()	このコマンドは、aci_gap_pass_key_req_event に対応してホストから送信される必要があります。 コマンド・パラメータには、固定ピンがない場合にペアリング処理中に使用されるパスキーが含まれています。
aci_gap_numeric_comparison_value_confirm_yesno()	ユーザはこのコマンドによって、aci_gap_numeric_comparison_value_event 経由で示された数値比較値を確認することもしないことも可能です。 デバイスがボンディングされている場合、キーは不揮発性領域に格納されます（Bluetooth LE リンク切断後）。つまり、デバイスがかつてボンディングされていた場合で、デバイスの 1つが電源を抜かれても、キーは失われません。 force_rebond パラメータを no force rebond にセットして aci_gap_set_pairing_req() コマンドが送信された場合、ペアリングは他の交換を行わずに完了します。
セキュリティ・データベースをクリアするには、以下のコマンドを使用します。	
aci_gap_clear_security_db()	セキュリティ・データベース内のすべてのデバイスは削除されます。

6.6.3 セキュリティ情報コマンド

表 10. セキュリティ情報コマンド

コマンド	説明
aci_gap_get_bonded_devices()	このコマンドは、ボンディングされているデバイスのリストを取得します。アドレス数と、対応するアドレスタイプおよび値を返します。
aci_gap_is_device_bonded()	このコマンドは、コマンドでアドレスが指定されたデバイスがボンディングされているかどうかを求めます。デバイスがリゾルブ可能プライベートアドレスを使用していて、かつボンディングされている場合、このコマンドは ble_status_successを返します。
aci_gap_get_security_level()	このコマンドは、デバイスの現在のセキュリティ設定要件を取得するために使用できます。
キー押し通知がサポートされている場合には、以下のコマンドを使用します。	
aci_gap_passkey_input()	このコマンドによって、パスキー入力中に検出された入力タイプをスタックに伝えることが可能となります。
OOB がサポートされている場合には、以下のコマンドを使用します。	
aci_gap_set_oob_data()	このコマンドは、OOB 通信を経由して届いた OOB データを入力するためにユーザによって送信されます。
aci_gap_get_oob_data()	このコマンドは、スタック自体によって生成された OOB データを取得（スタックから抽出）するためにユーザによって送信されます。

6.7 プライバシー機能

Bluetooth LE プライバシー機能によって、デバイスアドレスを頻繁に変更することにより、一定時間デバイスを追跡する能力が抑制されます。

プライバシー・モードを使用しているデバイスのアドレスは、
ペアリング処理中に交換される暗号化キーの 1つである IRK（ID 解決キー）を使用することで解決可能です。

デバイスは、まずプライバシーを無効にして初期化した後に接続とペアリングを行う必要があります。

その後で、両方のプライバシーを有効化するために、両側に次のコマンドを送信します。

`Hci_reset()`

`aci_gap_init()` - プライバシー有効

`aci_gap_add_devices_to_resolving_list()`

このコマンドは、コントローラ内のリゾルブ可能プライベートアドレスを解決するために使用されるアドレス変換リストに、デバイスを 1台追加するために使用されます。

セントラル側から次のコマンドを送信します。

`aci_gap_create_connection()`

または

`aci_gap_start_auto_connection_establish_proc()`

または

`aci_gap_start_general_connection_establish_proc()` (then `aci_gap_create_connection)` : `own_address_type` = リゾルブ可能プライベートアドレス, `peer_address_type` = パブリックまたはランダム

ペリフェラル側から次のコマンドを送信します。

`aci_gap_set_discoverable()`

または

`aci_gap_set_direct_connectable()`

または

`aci_gap_set_undirected_connectable()`

`own_address_type` = リゾルブ可能プライベートアドレス

接続が確立されると、LE 拡張接続完了イベントが生成されます。

6.8 2 Mbps 機能の使用方法

デバイス初期化フェーズにおいて、次のコマンドで好みの TX_PHYS と RX_PHYS の値を初期化できます。

表 11. 2 Mbps 機能コマンド

コマンド	説明
HCI_LE_Set_default_Phy()	接続にリンクされていない、(RX と TX に対する) 推奨 PHY の実行を指定します。デフォルトでは、推奨 PHY は 2M です。
接続が確立 (1M) されたら、TX と RX に対する推奨 PHY を互いに送信できます。	
HCI_LE_Set_Phy()	接続に対する推奨値をホストが指定できるようになります。
接続中は、使用されている TX と RX の PHY を読み出すことができます。	
HCI_LE_Read_Phy()	接続のための現在の PHY TX と RX を読み出します。

コマンド HCI_LE_Set_Phy() が使用されると、マスタはイベント hci_le_phy_update_complete を受信します。

6.9 接続パラメータの更新方法

接続が確立されると、接続パラメータの更新が可能となります。

表 12. 独自接続データ

コマンド	説明
マスタデバイス (セントラル) が更新のイニシエータである場合	
aci_gap_start_connection_update()	接続の更新を開始します (役割がマスタである場合のみ)。手続きが完了すると、上位レイヤに HCI_LE_CONNECTION_UPDATE_COMPLETE_EVENT イベントが返されます。
スレーブデバイス (ペリフェラル) が更新のイニシエータである場合	
aci_l2cap_connection_parameter_update_req()	スレーブからマスタに L2CAP 接続パラメータ更新リクエストを送信します。マスタがリクエストに応答 (許可または拒否) を返すと、HCI_L2CAP_CONNECTION_UPDATE_RESP_EVENT イベントが立てられます。

6.10 ロング・ローカル値またはロング・ディスタント値の書込みと読出し

6.10.1 ロング・ディスタント値の書込み

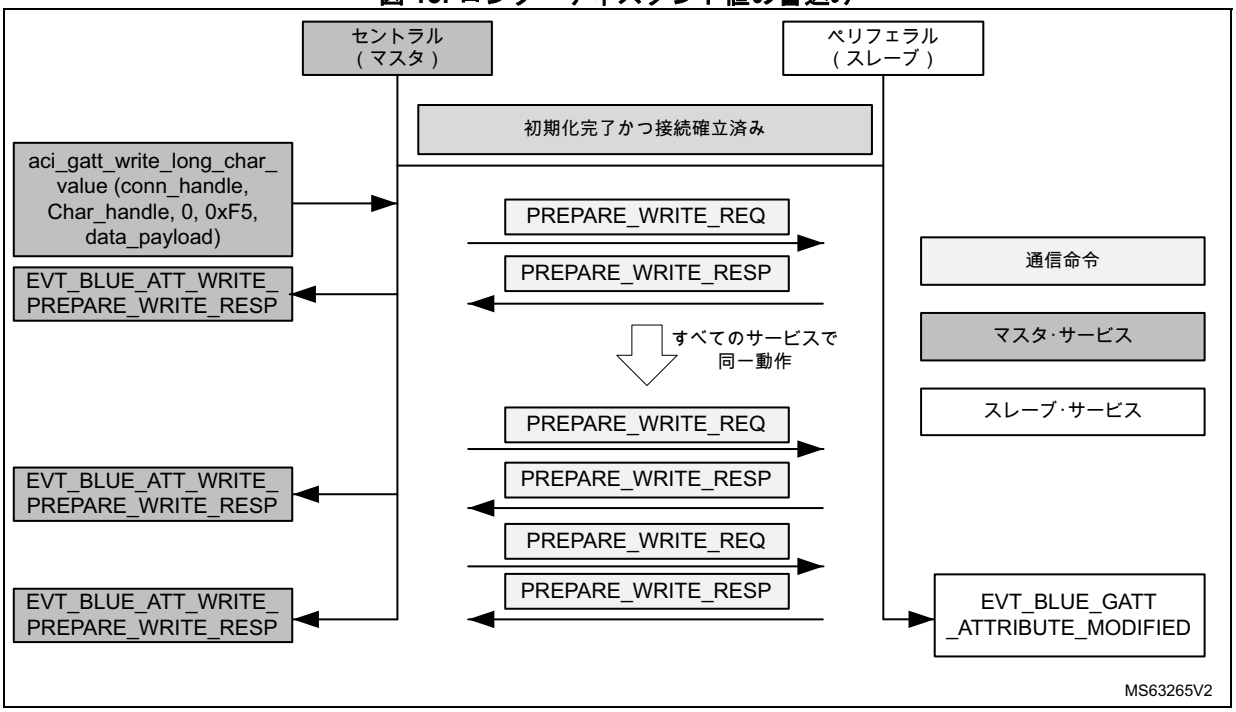
```
aci_gatt_write_long_char_value(conn_handle, RxCharHandle,offset,
chunk_size, (&payload)+offset)
```

最大データ書込み長は243バイトです。

サーバ側では、サービス aci_gatt_add_service を生成します。

図 13 に示すように、最大長 = 0xFF、プロパティを CHAR_PROP_WRITE として、キャラクタリスティック aci_gatt_add_char を生成します。

図 13. ロング・ディスタント値の書込み

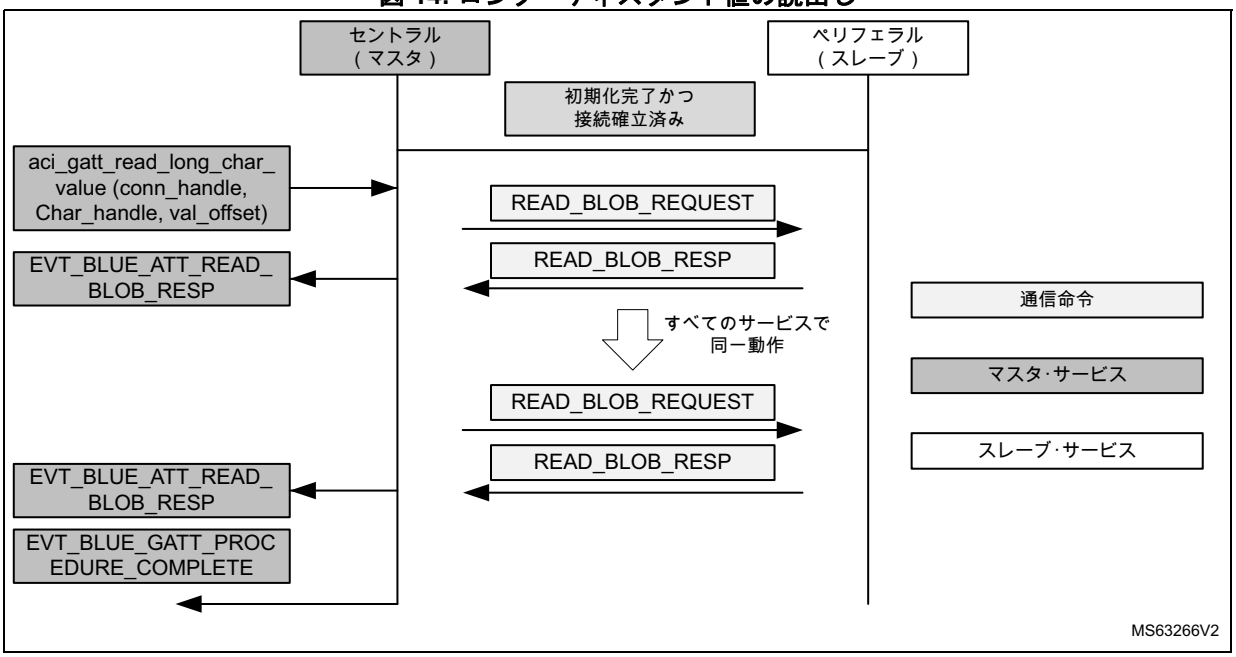


6.10.2 ロング・ディスタント値の読出し

```
aci_gatt_read_long_char_value(conn_handle, RxCharHandle, offset)
```

図 14 に示すように、最大データ読出し長は243バイトです。

図 14. ロング・ディスタント値の読出し



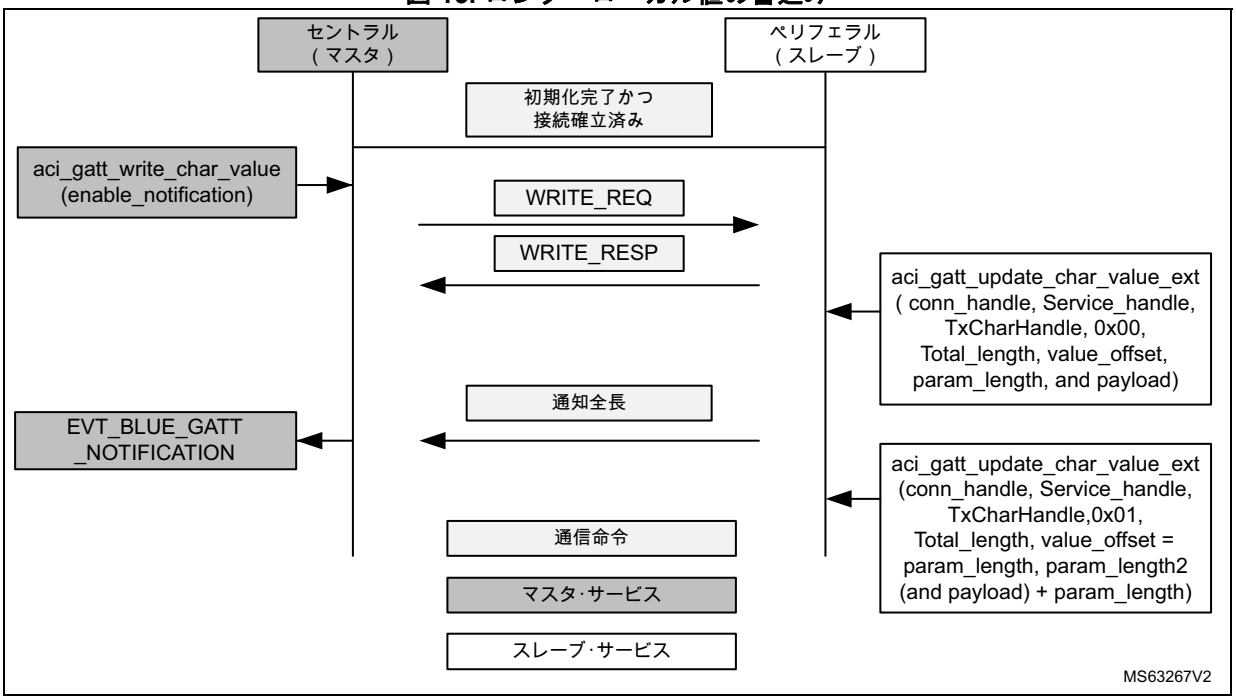
6.10.3 ロング・ローカル値の書込み

```
aci_gatt_update_char_value_ext(conn_handle_to_notify, service_handle,
char_handle, update_type, char_length, value_offset, value_length, value)
```

最大データ書込み長は 243バイトです。

図 15 に示すように、全長が（ATT_MTU – 3）を超えていなければすべてのデータが送信されます。

図 15. ロング・ローカル値の書込み

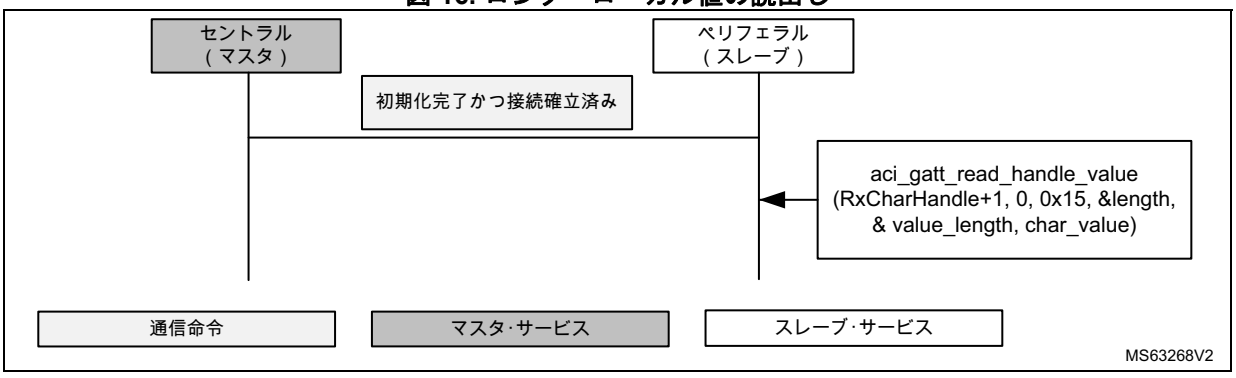


6.10.4 ロング・ローカル値の読出し

```
aci_gatt_read_handle_value(RxCharHandle+1, offset,
value_length_requested,&length,&value_length,&value)
```

1つのコマンドで243バイトを超えて読み出すことができない場合には、Value_length_requested は243を上回ることはできません。オフセットを使用する必要があります（図 16 参照）。

図 16. ロング・ローカル値の読出し



6.11 イベントとエラーコードの説明

スタック API がコールされた場合、API 戻りステータスを取得して、あらゆる潜在的エラー条件の監視と追跡を行います。

API が正常に実行されると、BLE_STATUS_SUCCESS（0x00）が返されます。

すべてのコマンド（(HCI - ACI）は、hci_command_status_event() による確認応答を受けます。

このコマンド・ステータス・イベントは、Command_Opcode パラメータによって記述されたコマンドが受信され、Bluetooth LE スタック・コントローラはこのコマンドに対するタスクを現在実行していることを示すために用いられます。

何か問題があると、Status イベント・パラメータには対応するエラーコードが含まれます（[7], v5.0, Vol. 2, part D 参照）。

GATT クライアント側では、いくつかの理由で GATT ディスカバリ手順が失敗する可能性があります。GATT サーバからのエラー応答の受信時に aci_gatt_error_resp_event() が生成されます。これは、手順がエラーで終わったことを意味しているのではなく、このエラーは手順自体の一部です。

すべての GATT クライアント手順は成功または aci_gatt_proc_complete_event() に対するエラーで完了していて、GATT クライアントが新しい手順を開始可能である必要があります。

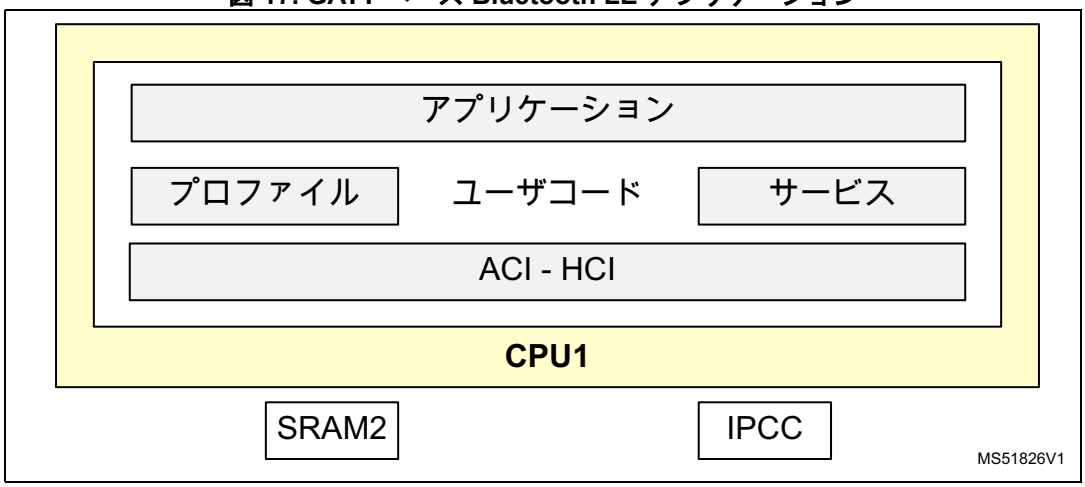
参考資料

7 BT-SIG と独自の GATT ベース Bluetooth LE アプリケーション

このセクションでは、CPU1で動作する次のアプリケーションの仕様と実装について説明します。

- STM 固有アプリケーション
  - Transparent mode - ダイレクトテストモード
- BT-SIG GATT ベース・アプリケーション
  - 心拍数センサ
- STMicroelectronics 独自 GATT ベース・アプリケーション
  - P2P アプリケーション（サーバ／クライアント）
  - FUOTA

図 17. GATT ベース Bluetooth LE アプリケーション



7.1 Transparent mode - ダイレクトテストモード（DTM）

7.1.1 目的と適用範囲

Bluetooth Specification Core v5.0 Low Energy Controller Volumeに記載されているように、一連の HCI コマンドの中には、ダイレクトテストモード（DTM）の有効化に使用されるコマンドのサブセットがあります。

DTM は DUT の制御に使用され、テスタにレポートを提供します。仕様に従って、次の 2つの手法のいずれかを使用して DTM をセットアップする必要があります。

1. HCI 上（STM32WB デバイスに実装されているもの）
2. 2線式 UART インタフェース経由

STM32WB は、Bluetooth Core Specification v5.0 [Vol. 6, Part F] に従って DTM をサポートしています。

以下は、仕様を完全に満たしている HCI テストコマンドです。

- HCI_LE_Transmitter_Test
- HCI_LE_Enhanced_Transmitter_Test
- HCI_LE_Receiver_Test
- HCI_LE_Enhanced_Receiver_Test
- HCI_LE_Test_End

受信テストパケット数が関数 HCI_LE_Test_End の戻り値です。

それ以外に使用可能な関数を表 13 に示します。

表 13. ダイレクトテストモード関数

機能	説明
aci_hal_le_tx_test_packet_number	このコマンドは、DTM テスト中に転送されたテストパケット数を返します。
aci_hal_set_tx_power_level	このコマンドは、TX 出力電力のセットに使用します。ACI コマンドセット（[3] 参照）に TX 電力レベル値の説明一式が含まれています。
aci_hal_tone_start	このコマンドは、STM32WB 無線からの連続波形（CW）の生成に使用されます。
aci_hal_tone_stop	このコマンドは、CW 放射の終了に使用されます。

以下のコマンドシーケンスは、STM32WB 無線からの CW 送信を有効化するための標準フローです。

```
hci_reset
aci_hal_set_tx_power_level
aci_hal_tone_start
aci_hal_tone_stop
```

7.1.2 Transparent mode アプリケーションの原理

このファームウェアは次の場合に使用されます。

- UART RX からコマンドを受信
- UART TX からイベントを送信
- IPCC 経由で Bluetooth LE スタックと通信

CPU1 アプリケーション·ファームウェアでの解釈は行われません。

1組のコマンド／イベントが transparent mode アプリケーションを介して Bluetooth LE スタックを制御するために STM32WB UART を通過する必要があります。

レベルシフタ、VCP ST-LINK、または適用可能な VCP を使用して TX と RX を管理することができます。

7.1.3 設定

STM32WB は2線式 UART インタフェース（TXD、RXD）として見えます。CPU1アプリケーションは、「Ble_TransparentMode」で使用されます。このファームウェアは、コマンドとイベントの解釈は行わずに、IPCC および選択された UART インタフェース（USART1 または LPUART1）経由でワイヤレス Bluetooth LE スタックとの通信のみを行います。

UART インタフェースと設定の選択は、app_conf.h を使用して行われます。

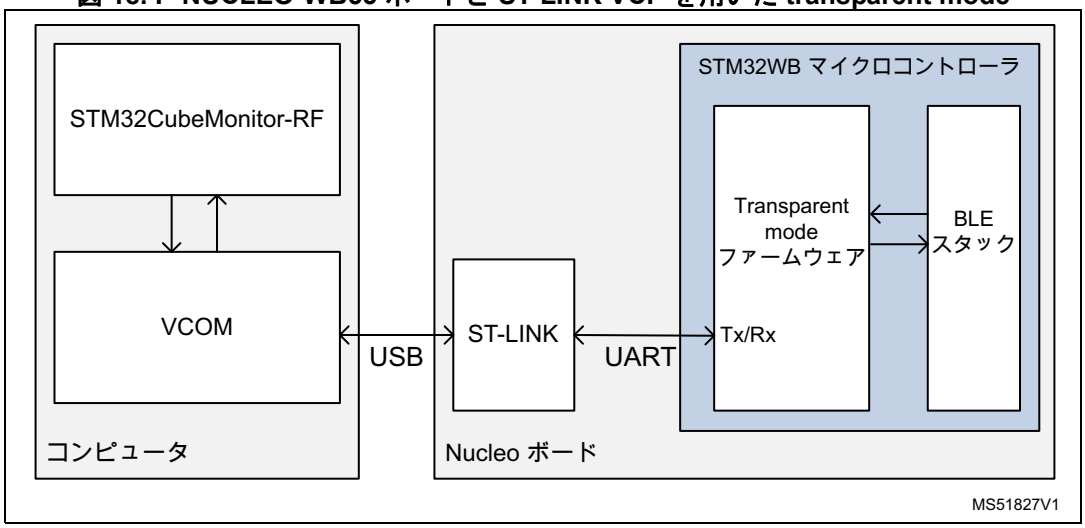
```
#define CFG_UART_GUI            hw_uart1
```

P-NUCLEO-WB55 ボードには、仮想 COM ポート機能を備えた ST-LINK が搭載されています。

次のプロジェクトは、ST-LINK の VCP 経由で通信するように設定されています。
\Projects\ NUCLEO-WB55.Nucleo\Applications\BLE\Ble_TransparentMode.

UART1（PB6、PB7）は、P-NUCLEO-WB55 ボードの ST-LINK VCP に接続されています。

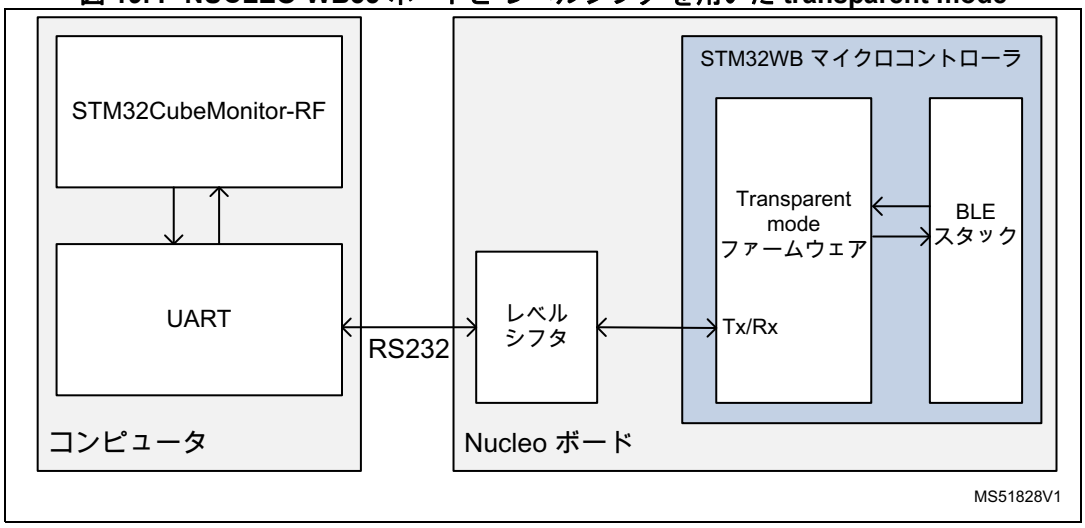
図 18. P-NUCLEO-WB55 ボードと ST-LINK VCP を用いた transparent mode



P-NUCLEO-WB55 ドングル･ボードは、ST-LINK なしで提供されます。プロジェクト .\Projects\NUCLEO-WB55.USBDongle\Applications\BLE\BLE_TransparentModeVCP には、transparent mode 機能に加えて仮想 COM ポートの実装が含まれています。

レベルシフタ（NUCLEO-WB55RG ボードには同梱されていません）を介して、STM3232WB UART インタフェースを直接 RS232 シリアル通信に接続することもできます。この手法は、いろいろある中でも RF テスタの接続に使用できます。

図 19. P-NUCLEO-WB55 ボードと レベルシフタ を用いた transparent mode

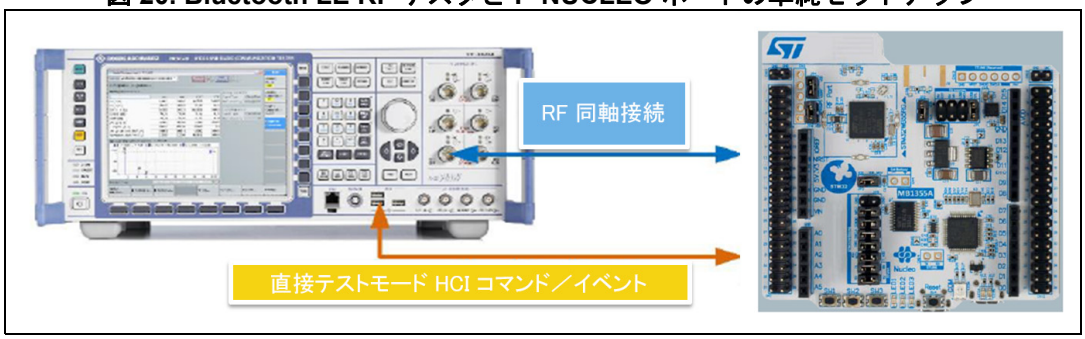


7.1.4 RF 認証 - アプリケーション実装

ダイレクトテストモード（DTM）は、USB または RS232 インタフェースに対するリモートコントロールコマンドなど、Bluetooth LE デバイスにさまざまな種類の RF テストを提供するように Bluetooth SIG によって規定されています。

RF 評価のため、Bluetooth LE RF は、変調ありまたはなしで、連続送信モードか連続受信モードに設定されます。図 20 に単純なセットアップ図を示します。

図 20. Bluetooth LE RF テスタと P-NUCLEO ボードの単純セットアップ

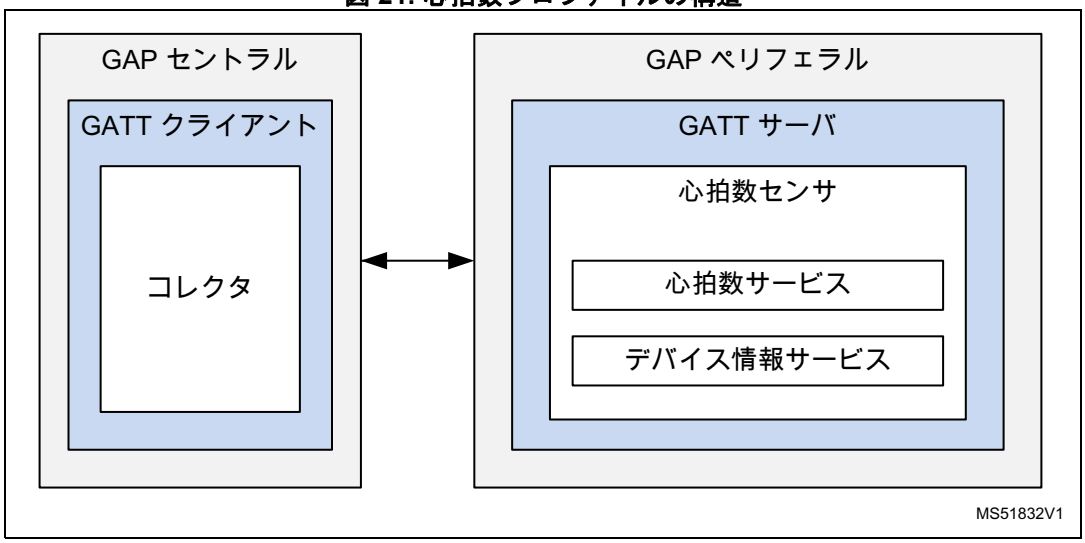


7.2 心拍数センサアプリケーション

心拍数プロファイルは次の 2 つの動作から構成されます。

- コレクタ : 心拍数測定値およびその他のデータを受信する GAP セントラルと GATT クライアント
- 心拍数センサ: 心拍数測定値およびその他のデータを提供する GAP ペリフェラルと GATT サーバ

図 21. 心拍数プロファイルの構造



The STM32WBCube_FW_WB_V1.0.0 リリースは、心拍数センサ例とともに提供されます。

このセクションでは、（フィットネス･アプリケーション用として）センサから心拍数を毎秒送信することを目的とした、次のステップから構成されている Bluetooth SIG 心拍数センサ･アプリケーションを作成する手順について説明します（図 22 参照）。

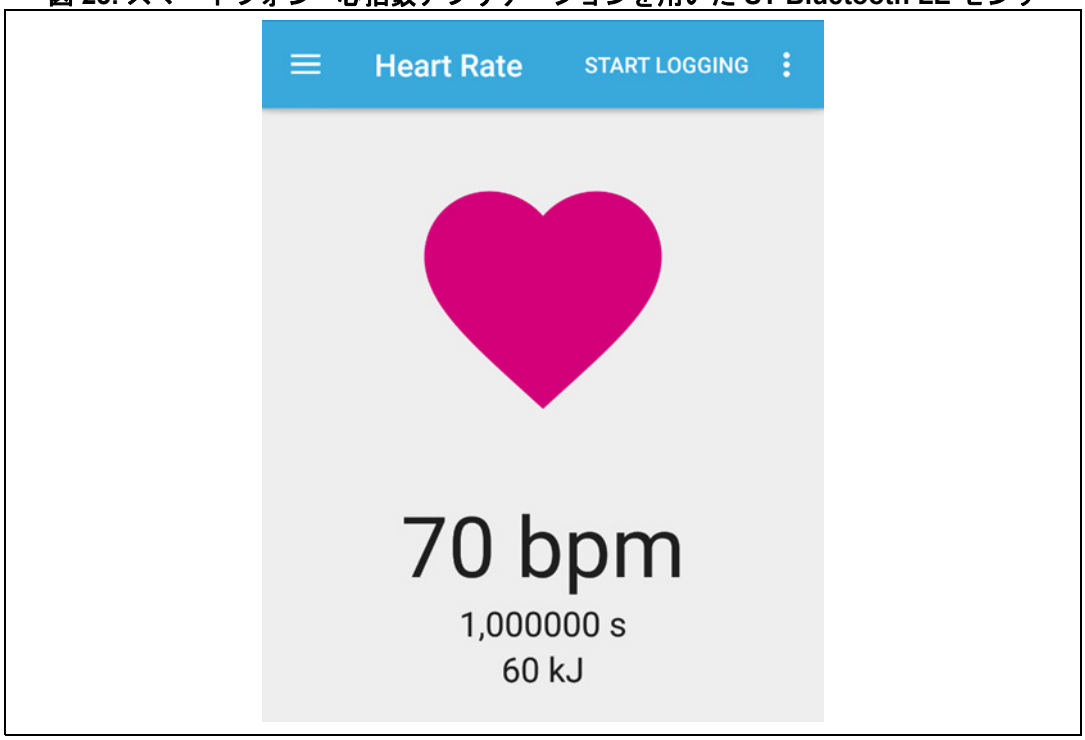
- STM32WB ユーザアプリケーションの初期化
- 心拍数サービスの実装 - ミドルウェア
- 心拍数センサペリフェラル - ユーザ
- 心拍数センサ測定値を更新 -ユーザ

図 22. Bluetooth LE RF テスタと P-NUCLEO ボードの単純セットアップ

7.2.1 STM32WB 心拍数センサ･アプリケーションの使用方法

- BLE_HeartRate プロジェクトを開いて readme.txt に書かれている指示に従います。
- ST Bluetooth LE センサ･モバイル･アプリケーションをお手持ちの心拍数アプリケーションに接続します。

図 23. スマートフォン - 心拍数アプリケーションを用いた ST Bluetooth LE センサ



7.2.2 STM32WB 心拍数センサ･アプリケーション - ミドルウェア･アプリケーション

Middlewares\STM32_WPAN\ble\core\Src\ において、Bluetooth LE サービスを挿入するサブフォルダは blesvc です。

警告： このフォルダのファイルは変更しないでください。

- svc_ctl.c : Bluetooth LE スタックを初期化して、アプリケーションのサービス（GATT イベント）を管理します。
- hrs.c : 次の目的に使用されます。
  - サービスとアプリケーションに対するそのキャラクタリスティックの生成
  - サービス･キャラクタリスティックの更新
  - 通知または書込みコマンドの受信
  - Bluetooth LE スタックとアプリケーションの間のリンクの作成

アプリケーションに対して、固有のコードを生成するサブフォルダは STM32_WPAN\app です。

- app_entry.c : Bluetooth LE トランスポートレイヤと BSP（LED、ボタンなど）を初期化します。
- app_ble.c : GAP を初期化して、接続（アドバタイジング、スキャンなど）を管理します。
- hrs_app.c : GATT を初期化して、アプリケーションを管理します。

心拍数サービスの機能：Middlewares\STM32_WPAN\ble\core\Src\blesvc\hrs.c

表 14. 心拍数サービスの機能

機能	説明
`Service Init - HRS_Init()`	– 心拍数イベントハンドルをサービス･コントローラに登録します。 – サービス UUID を初期化します。
`aci_gatt_add_serv`	– 心拍数サービスをプライマリ・サービスとして追加します。 – 心拍数測定キャラクタリスティックを初期化します。
`aci_gatt_add_char`	– 心拍数キャラクタリスティックを追加します。 – ボディ・センサ位置キャラクタリスティックを初期化します。
`aci_gatt_add_char`	– ボディ・センサ位置キャラクタリスティックを追加します。 – 心拍数測定キャラクタリスティックを更新します。
`aci_gatt_update_char_value`	– 指定値内のリクエストされたキャラクタリスティックを更新します。 – ボディ・センサ位置キャラクタリスティック値を更新します。
`aci_gatt_update_char_value`	– 指定値内のリクエストされたキャラクタリスティックを更新します。
`HeartRate_Event_Handler(void *Event)`	– HCI Vendor Type Event を管理します。
`EVT_BLUE_GATT_WRITE_PERMIT_REQ`	– サーバが書込みコマンドを受信します。 – HR 制御点キャラクタリスティック値 – 電力拡張コマンドをリセットし、その後、 OK ステータスで aci_gatt_write_response() を送信します。 拡張された電力をリセットするように HRS アプリケーションに通知します。 または、エラーで aci_gatt_write_response() を送信します。
`EVT_BLUE_GATT_ATTRIBUTE_MODIFIED`	– HR 測定キャラクタリスティック記述値 – 通知の有効化または無効化 – 測定通知を HRS アプリケーションに通知します。

サービス実装の目的は次のとおりです。

- 心拍数サービスと選択されたキャラクタリスティックを Bluetooth LE スタック GATT データベースに登録します。

```
/**
 * @brief  Service Heart Rate initialization
 * @param  None
 * @retval None
 */
void HRS_Init(void)
{
```

心拍数イベントハンドラを登録

- `SVCCTL_RegisterSvcHandler(HearRate_Event_Handler);`

心拍数サービス GATT データベースを Bluetooth LE スタックに登録
Add Heart Rate Service
Add Heart Rate characteristics
Measurement Value (mandatory)
Body sensor Location (Optional)
Heart rate control point(Optional)
Add Over The Air Reboot Request characteristic (Optional)
}

- HR サービス専用の GATT イベントを管理します。

```
/**
 * @brief Heart Rate Service Event handler
 * @param Event: Address of the buffer holding the Event
 * @retval Ack: Return whether the GATT Event has been managed or not
 */
static SVCCTL_EvtAckStatus_t HearRate_Event_Handler(void *Event)
{
```

Bluetooth LE スタックからの GATT イベントを管理
- EVT_BLUE_GATT_WRITE_PERMIT_REQ
- EVT_BLUE_GATT_ATTRIBUTE_MODIFIED

ユーザアプリケーションに通知 - HRS_Notification
- HRS_RESET_ENERGY_EXPENDED_EVT
- HRS_NOTIFICATION_ENABLED
- HRS_NOTIFICATION_DISABLED
- HRS_STM_BOOT_REQUEST_EVT

}

- Bluetooth LE スタック GATT データベースに対するキャラクタリスティックのアプリケーションによる更新を可能とします。

```
/**
 * @brief  Characteristic update
 * @param  UUID: UUID of the characteristic
 * @retval BodySensorLocationValue: The new value to be written
 */
tBleStatus HRS_UpdateChar(uint16_t UUID, uint8_t *pPayload)
{
```

ボディ・センサ位置を更新

心拍数測定値を更新
}

サービス･コントローラの機能: Middlewares\STM32_WPAN\ble\core\Src\blesvc\svc_ctl.c

The SVCCTL_Init() にはさまざまな機能があります。

- すべての開発されたサービスの初期化関数をコールします。
  - HR サーバ - HRS_Init()
- サービスイベントハンドラを登録します。
  - SVCCTL_RegisterSvcHandler()
  - svc_ctl.c から GATT イベントを受信してアプリケーションにリダイレクトする関数（hrs_app.c）
- クライアントイベントハンドラを登録します（HR センサ・プロジェクトには適用されません）。
  - SVCCTL_RegisterCltHandler()

HR センサ・アプリケーションの初期化: Applications\BLE\BLE_HeartRate\STM32_WPAN\App\app_ble.c

HR センサ・ペリフェラルの初期化 - APP_BLE_Init()

- CPU2 の Bluetooth LE スタックを初期化します。
  - SHCI_C2_BLE_Init()
- HCI、GATT、GAP レイヤを初期化します。
  - Ble_Hci_Gap_Gatt_Init()
- Bluetooth LE サービスを初期化します。
  - SVCCTL_Init()
- 心拍数サーバとデバイス情報アプリケーションの初期化をコールします。
  - HRSAPP_Init()
  - DISAPP_Init()
- ADV パラメータ、ローカル・ネーム、UUID などを設定してアドバタイジングを開始します。
  - aci_gap_set_discoverable() - デバイスを一般ディスカバリ可能モードに設定します。
  - aci_gap_update_adv_data() - アドバタイジング・データ・パケットに情報を追加します。
- GAP Event - SVCCTL_App_Notification() を管理します。
  - EVT_LE_CONN_COMPLETE

接続間隔情報、スレーブ遅延、監視タイムアウトを提供します。

- 接続の新しい情報を提供します。
  - EVT_LE_CONN_UPDATE_COMPLETE
- リンクの切断とその理由をアプリケーションに通知します。
  - EVT_DISCONN_COMPLETE
- リンクが暗号化されているかどうかをアプリケーションに通知します。
  - EVT_ENCRYPT_CHANGE

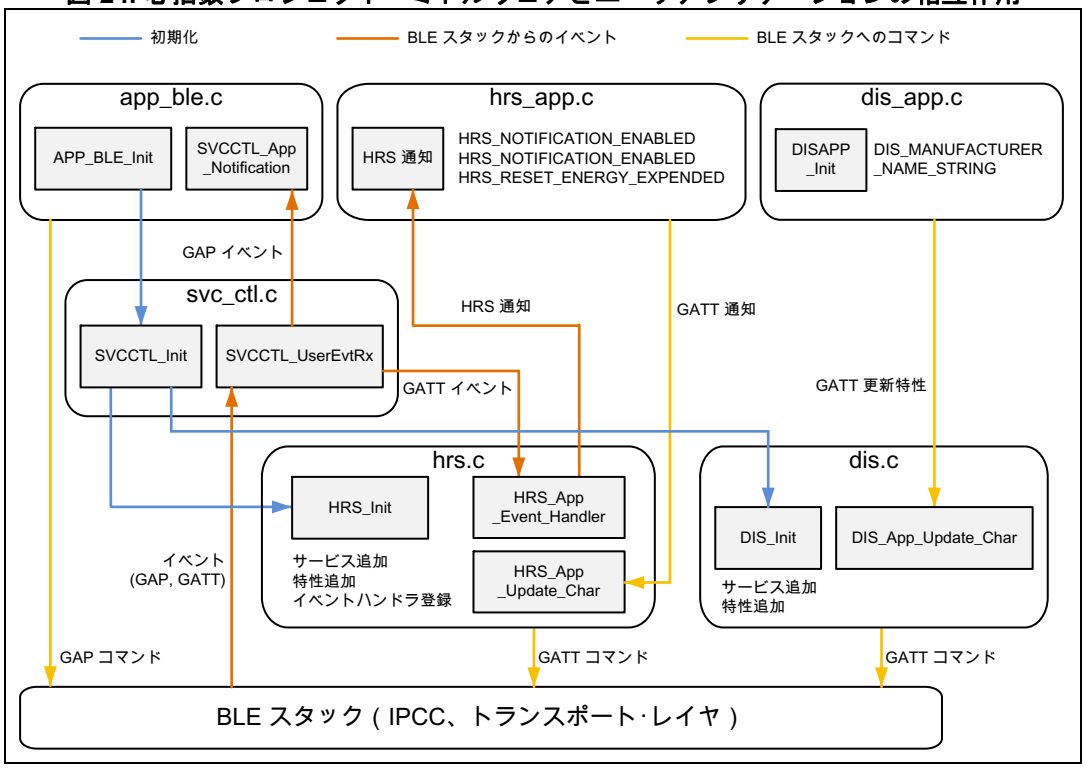
HR センサ・アプリケーションの制御: Applications\BLE\BLE_HeartRate\STM32_WPAN\App\hrs_app.c

hrs_app.c ファイルは、センサ・アプリケーションを初期化してタイマを生成します。

表 15. HR センサ・アプリケーションの制御

関数	説明
HRSAPP_Init()	GATT レベルで Bluetooth LE スタックからの内部イベントを受信して処理します。
HRS_Notification()	サービス関数をコールしてキャラクタリスティックを更新します(通知／書込み。
HRSAPP_Measurement()	-

図 24. 心拍数プロジェクト - ミドルウェアとユーザアプリケーションの相互作用



7.3 STMicroelectronics 独自アドバタイジング

デバイスがペリフェラルである場合、Bluetooth アドレス、アドバタイジング・ペイロード（0～31バイト長）などの情報をアドバタイズします。

アドバタイジング情報は、Bluetooth SIG で標準化されているアドバタイジング・データ・エレメントによって表されます。

- 第 1バイト：エレメント長（長さバイト自体を除く）
- 第 2バイト：AD タイプ - エレメントにどのデータが含まれるかを指定します。
- AD データ：AD タイプによってその意味が定義される 1バイト以上

AD タイプの 0xFF は、製造業者専用データの提供に使用されます。

P2P アプリケーションと FUOTA アプリケーションのような STMicroelectronics 独自 GATT ベース・アプリケーションの実装は、製造業者専用 AD タイプデータで提案されます。これは、リモートデバイス（スキャナ）がペリフェラルデバイスを選別して、リクエストされたアプリケーションにアクセスする方法です。

表 16. Bluetooth 5 Core Specification Vol. 3 part C による AD 構造

フィールド名	タイプ	長さ	レコード・サイズ
TX_POWER_LEVEL	0x0A	2	3
COMPLETE_NAME	0x09	8	9
MANUF_SPECIFIC	0xFF	13	14
FLAGS	0x01	2	3

表 17. STM32WB 製造業者専用データ

オクテット	0	1	2	3	4	5	6	7	8	9	10	11	12	13
名前	長さ	タイプ	バージョン	DevID	グループ A 機能		グループ B 機能		パブリックデバイスアドレス（48ビット）（オプション）					
値	0x0D	0xFF	0x01	0xXX	RFU		0xXXXX		0xXXXXXXXXXX					

グループ B 機能

- ビットマスク Thread : Thread スイッチ・キャラクタリスティックの存在のアドバタイズに使用されます。
- ビットマスク OTA リブート・リクエスト : Bluetooth LE リブート・キャラクタリスティックの存在のアドバタイズに使用されます。

表 18. グループ B 機能 - ビットマスク

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFU	Thread サポート	OTA リブート・リクエスト	RFU												

表 19. デバイス ID 列挙型

ID	HW
0x00	汎用
0x83	STM32WB P2P サーバ 1
0x84	STM32WB P2P サーバ 2
0x85	STM32WB P2P ルータ
0x86	STM32WB FUOTA

アドバタイジング手続きは、app_ble.c の中でユーザアプリケーションレベルで管理されます。

以下は、BLE_p2p サーバプロジェクトのアドバタイジング開始の例です。

```
/* Local name to be advertised */
static const char local_name[ ] = { AD_TYPE_COMPLETE_LOCAL_NAME, 'P', '2',
'P', 'S', 'R', 'V', '1' };

/* manufacturer data & legacy data to be advertised */
uint8_t manuf_data[14] = {
```

```
    sizeof(manuf_data)-1, AD_TYPE_MANUFACTURER_SPECIFIC_DATA,
    0x01/* SKD version */,
    CFG_DEV_ID_P2P_SERVER1 /* STM32WB - P2P Server 1*/,
    0x00 /* GROUP A Feature  */,
    0x00 /* GROUP A Feature */,
    0x00 /* GROUP B Feature */、
    0x00 /* GROUP B Feature */、
    0x00, /* BLE MAC start - MSB */
    0x00
    0x00
    0x00
    0x00
    0x00, /* BLE MAC stop */
};

/* Local device BD address */
const uint8_t *bd_addr;
bd_addr = SVCCTL_GetBdAddress();

/* BLE MAC update for Advertising manufacturer data */
manuf_data[ sizeof(manuf_data)-6] = bd_addr[5];
manuf_data[ sizeof(manuf_data)-5] = bd_addr[4];
manuf_data[ sizeof(manuf_data)-4] = bd_addr[3];
manuf_data[ sizeof(manuf_data)-3] = bd_addr[2];
manuf_data[ sizeof(manuf_data)-2] = bd_addr[1];
manuf_data[ sizeof(manuf_data)-1] = bd_addr[0];

/* Put the GAP peripheral in general discoverable mode:
Advertising_Type: ADV_IND(undirected scannable and connectable);
Advertising_Interval_Min;
Advertising_Interval_Max;
Own_Address_Type: PUBLIC_ADDR (public address: 0x00);
Adv_Filter_Policy: NO_WHITE_LIST_USE (no whit list is used);
Local_Name_Length
Local_Name:
Service_Uuid_Length: 0 (no service to be advertised);
Service_Uuid_List: NULL;
Slave_Conn_Interval_Min: 0 (Slave connection internal minimum value);
Slave_Conn_Interval_Max: 0 (Slave connection internal maximum value).
*/
result = aci_gap_set_discoverable(ADV_IND,
CFG_FAST_CONN_ADV_INTERVAL_MIN,
CFG_FAST_CONN_ADV_INTERVAL_MAX,
PUBLIC_ADDR,
NO_WHITE_LIST_USE, /* use white list */
```

参考資料


```
sizeof(local_name), (uint8_t*) local_name,
0,
NULL,
0, 0);

 /* Update Advertising data with manufacturer specific information */
result = aci_gap_update_adv_data(sizeof(manuf_data), (uint8_t*)
manuf_data);
```

結果は、常に BLE_STATUS_SUCCESS（0x00）と比較されます。

7.4 独自 P2P アプリケーション

以下の 3つのコンポーネントを使用すれば、さまざまなデータ通信タイプのデモンストレーションが可能です。

1. P2P サーバ・プロジェクト
2. P2P クライアント・プロジェクト
3. スマートフォン・アプリケーション

異なるコンポーネントを組み合わせると、図 25 および図 26 に示したデモンストレーションとなります。

図 25. P2P サーバツークライアント・デモンストレーション

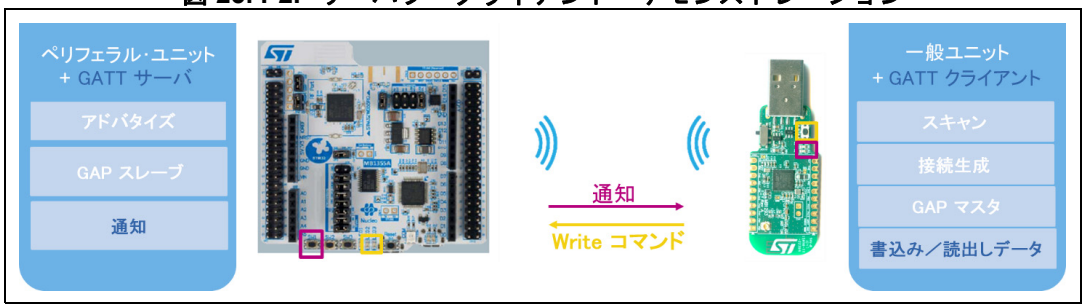
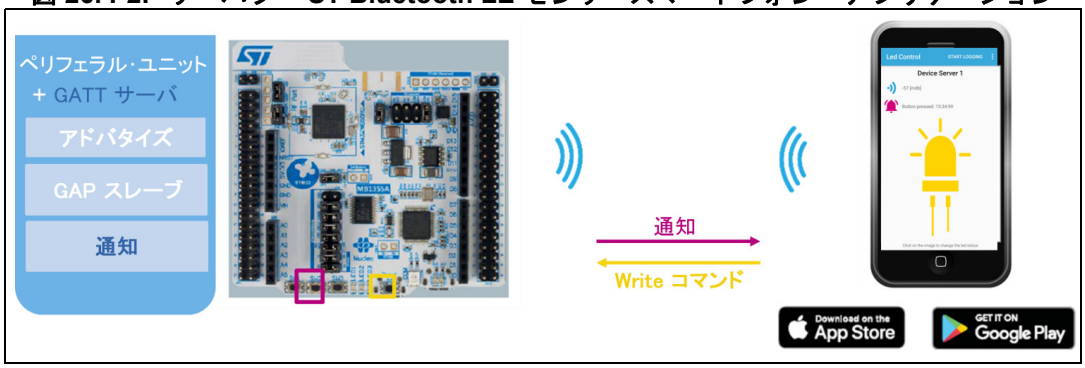


図 26. P2P サーバツー ST Bluetooth LE センサ・スマートフォン・アプリケーション



7.4.1 P2P サーバ仕様

ポイントツーポイント通信のデモンストレーションには、P2P サーバ・アプリケーションを使用する必要があります。次の GATT サービスとキャラクタリスティックを備えるペリフェラルデバイスとして機能します。

表 20. P2P サービスとキャラクタリスティック UUID

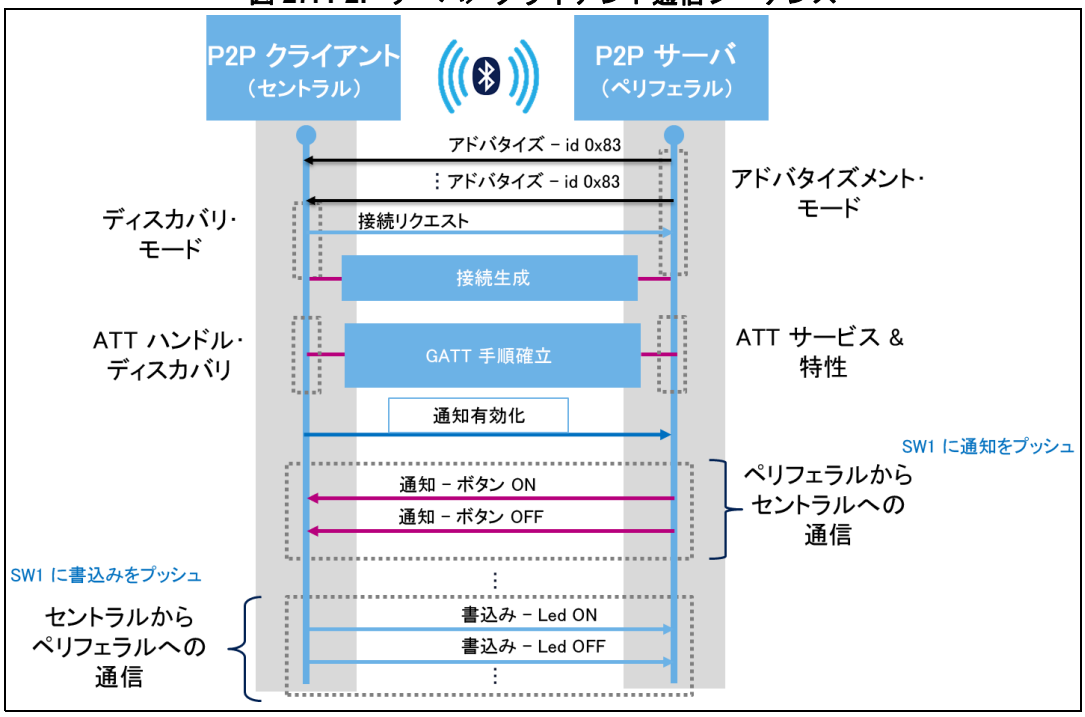
グループ	サービス	特徴	サイズ	モード	UUID
LED ボタン制御	P2P サービス	-	-	-	0000FE40-cc7a-482a-984a-7fed5b3e58f
	-	書込み	2	読出し／書込み	0000FE41-8e22-4541-9d4c-21edae82ed19
	-	通知	2	通知	0000FE42-8e22-4541-94dc-21edae82ed19

表 21. P2P 仕様

書込み	オクテット LSB	0	1
	名前	デバイス選択	LED 制御
	値	– 0x01：P2P サーバ 1 – 0x02：P2P サーバ 2 – 0x0x：P2P サーバ x – 0x00：すべて	– 0x00 LED オフ – 0x01 LED オン – 0x02 Thread スイッチ
通知	オクテット LSB	0	1
	名前	デバイス選択	ボタン
	値	– 0x01：P2P サーバ 1 – 0x02：P2P サーバ 2 – 0x0x：P2P サーバ x	– 0x00 スイッチ オフ – 0x01 スイッチ オン

使用するためには、GAP セントラルと GATT クライアント・デバイスは P2P サーバ・アプリケーションをディスカバーして接続する必要があります。図 27 にデータ交換手順が説明されています。

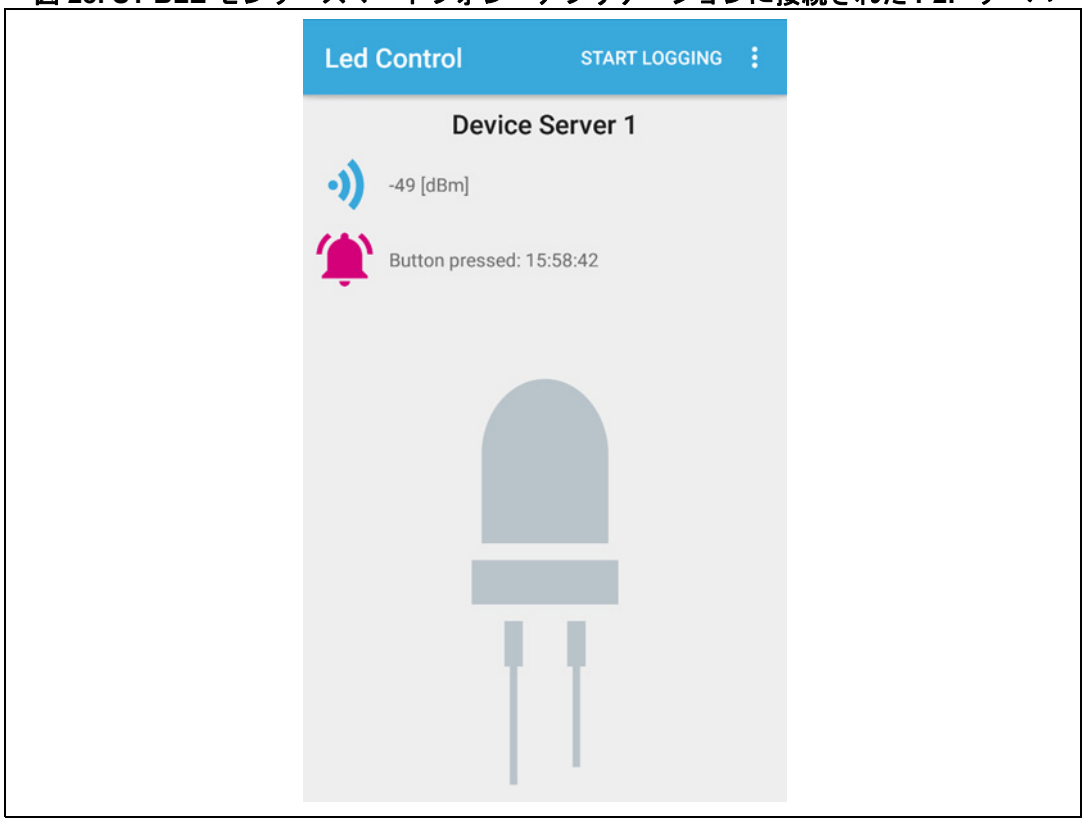
図 27. P2P サーバ／クライアント通信シーケンス



7.4.2 P2P サーバ・アプリケーションの使用方法

1. P-NUCLEO-WB55 ボードに the ble_P2P_Server プロジェクトを Flash 書込みします。
2. Flash 書込みを行ったら、ボードにST Bluetooth LE センサ·モバイル·アプリケーションを接続し、SW1 ボタンを使用してスマートフォンに通知を送ります。

図 28. ST BLE センサ・スマートフォン・アプリケーションに接続された P2P サーバ



7.4.3 P2P サーバ・アプリケーション - ミドルウェア·アプリケーション

P2P サービスとキャラクタリスティックは、p2p_stm.c ファイルを使用して生成されます。

p2p_stm.c : アプリケーションの中にサービスとキャラクタリスティックを生成し、通知を受信するかコマンドを書き込むために、キャラクタリスティックを更新します。さらに、Bluetooth LE ワイヤレス·スタックと適用可能部分の間にリンクを張ります。

アプリケーションの中で、固有のコードを生成するサブフォルダは「User」です。

- app_entry.c : Bluetooth LE トランスポートレイヤと BSP（LED、ボタンなど）を初期化します。
- app_ble.c : GAP を初期化して、接続（アドバタイジング、スキャンなど）を管理します。
- p2p_server_app.c : GATT を初期化して、アプリケーションを管理します。

P2P サービスの機能: Middlewares\STM32_WPAN\ble\core\Src\blesvc\p2p_stm.c

表 22. P2P サービスの機能

機能	説明
`Service Init - P2PS_STM_Init ()`	– サービス･コントローラに PeerToPeer_Event_Handler を登録します。 – サービス UUID を初期化します。 aci_gatt_add_serv - P2P サービスをプライマリ・サービスとして追加します。 – P2P 書込みキャラクタリスティックを初期化します。 aci_gatt_add_char - 書込みキャラクタリスティックを追加します。 – P2P 通知キャラクタリスティックを初期化します。 aci_gatt_add_char - 通知キャラクタリスティックを追加します。 – 通知キャラクタリスティックを更新します。- P2PS_STM_App_Update_Char() aci_gatt_update_char_value - 仕様に沿った値で通知キャラクタリスティックを更新します。
`PeerToPeer_Event_Handler (void *Event) - manages HCI Vendor Type Event:`	EVT_BLUE_GATT_ATTRIBUTE_MODIFIED – 通知キャラクタリスティックのディスクリプタ値の設定を受信します。 – 通知の有効化または無効化 – P2PS_STM__NOTIFY_ENABLED_EVT または P2PS_STM_NOTIFY_DISABLED_EVT をアプリケーションに通知します。 – 書込みキャラクタリスティックに対するデータを受信し、P2P アプリケーションに通知します。 P2PS_STM_WRITE_EVT – リブート・リクエスト・キャラクタリスティックに対するデータを受信し、（FUOTA 手順に使用される）P2P アプリケーションに通知します。 P2PS_STM_BOOT_REQUEST_EVT

P2P サーバ・アプリケーションの制御： Applications\BLE\BLE_p2pServer\STM32_WPAN\App\p2p_server_app.c

p2p_server_app.c ファイルは以下の処理を行います。

P2P サーバ・アプリケーションを初期化して、タイマを生成します。

```
P2PS_APP_Init()
```

GATT レベルで Bluetooth LE スタックからの内部イベントを受信して処理します。

```
P2PS_STM_App_Notification ()


void P2PS_STM_App_Notification(P2PS_STM_App_Notification_evt_t
*pNotification)
{
Switch (pNotification->P2P_Evt_Opcode)
{
 case P2PS_STM__NOTIFY_ENABLED_EVT:
  P2P_Server_App_Context.Notification_Status = 1;
  APP_DBG_MSG("-- P2P APPLICATION SERVER : NOTIFICATION ENABLED\n");
  APP_DBG_MSG(" \n\r");
 break;
```

```
 case P2PS_STM_NOTIFY_DISABLED_EVT:
  P2P_Server_App_Context.Notification_Status = 0;
  APP_DBG_MSG("-- P2P APPLICATION SERVER : NOTIFICATION DISABLED\n");
  APP_DBG_MSG(" \n\r");
 break;

 case P2PS_STM_WRITE_EVT:
  if(pNotification->DataTransfered.pPayload[0] == 0x00){
  if(pNotification->DataTransfered.pPayload[1] == 0x01)
   {
    BSP_LED_On(LED_BLUE);
    APP_DBG_MSG("-- P2P APPLICATION SERVER  : LED1 ON\n");
    APP_DBG_MSG(" \n\r");
    P2P_Server_App_Context.LedControl.Led1=0x01;
    }
   if(pNotification->DataTransfered.pPayload[1] == 0x00)
   {
    BSP_LED_Off(LED_BLUE);
    APP_DBG_MSG("-- P2P APPLICATION SERVER  : LED1 OFF\n");
    APP_DBG_MSG(" \n\r");
    P2P_Server_App_Context.LedControl.Led1=0x00;
    }
   }
```

サービス関数をコールしてキャラクタリスティックを更新します（通知）。

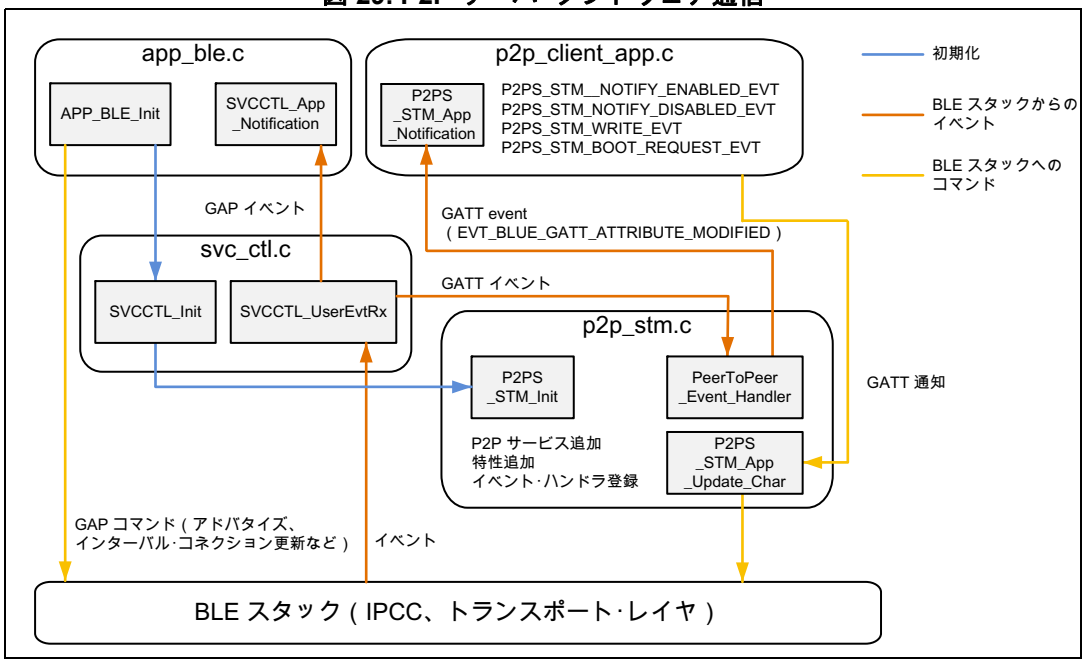
```
P2PS_Send_Notification ()
void P2PS_Send_Notification(void)
{
 if(P2P_Server_App_Context.ButtonControl.ButtonStatus == 0x00){
  P2P_Server_App_Context.ButtonControl.ButtonStatus=0x01;
 } else {
  P2P_Server_App_Context.ButtonControl.ButtonStatus=0x00;
 }

 if(P2P_Server_App_Context.Notification_Status){
  APP_DBG_MSG("P2P APPLICATION SERVER  : INFORM CLIENT BUTTON 1 PUSHED \n
");
  APP_DBG_MSG(" \n\r");
  P2PS_STM_App_Update_Char(P2P_NOTIFY_CHAR_UUID, (uint8_t *)
&P2P_Server_App_Context.ButtonControl);
  } else {
  APP_DBG_MSG("P2P APPLICATION SERVER : CAN'T INFORM CLIENT -  NOTIFICATION
DISABLED\n ");
  }
```

```
return;
}
```

7.4.4 P2P クライアント・アプリケーション - ミドルウェア·アプリケーション

図 29. P2P サーバ·ソフトウェア通信



P2P クライアントのために生成されるサービスはありません。アプリケーションレベルであらゆる GATT イベントについて通知を受ける GATT クライアント·ハンドラ SVCCTL_RegisterCltHandler() の登録のみが必要です。

アプリケーションの中で、固有のコードを生成するサブフォルダは「User」です。

- app_entry.c : Bluetooth LE トランスポートレイヤと BSP（LED、ボタンなど）を初期化します。
- app_ble.c : GAP を初期化して、接続（スキャンと接続）を管理します。
- p2p_client_app.c : GATT を初期化して、GATT クライアンアプリケーションを管理します。

P2P クライアント - スキャンと接続

app_ble.c ファイルは以下の処理を行います。

- スキャンを実行して P2P サーバ・アドバタイジング ID があるか探します。

```
static void Scan_Request( void )
{
 tBleStatus result;
 if (BleApplicationContext.Device_Connection_Status !=
APP_BLE_CONNECTED_CLIENT)
 {
  BSP_LED_On(LED_BLUE);
  result = aci_gap_start_general_discovery_proc(SCAN_P, SCAN_L,
PUBLIC_ADDR, 1);
```

```
  if (result == BLE_STATUS_SUCCESS)
  {
   APP_DBG_MSG(" \r\n\r** START GENERAL DISCOVERY (SCAN) **  \r\n\r");
  } else {
  APP_DBG_MSG("-- BLE_App_Start_Limited_Disc_Req, Failed \r\n\r");
  BSP_LED_On(LED_RED);
  }
 }
return;
}
```

- 選別される ADV イベントレポートを受信して P2P サーバ BD アドレス J'rric を保存します。

```
case AD_TYPE_MANUFACTURER_SPECIFIC_DATA: // Manufactureur Specific
 if (adlength >= 7 && le_advertising_event->Advertising_Report[0].Data[k +
2] == 0x01) {
  APP_DBG_MSG("--- ST MANUFACTURER ID --- \n");
  switch (le_advertising_event->Advertising_Report[0].Data[k + 3]) {
   case CFG_DEV_ID_P2P_SERVER1:
    APP_DBG_MSG("-- SERVER DETECTED -- VIA MAN ID\n");
    BleApplicationContext.DeviceServerFound = 0x01;
    SERVER_REMOTE_BDADDR[0] = le_advertising_event-
>Advertising_Report[0].Address[0];
    SERVER_REMOTE_BDADDR[1] = le_advertising_event-
>Advertising_Report[0].Address[1];
    SERVER_REMOTE_BDADDR[2] = le_advertising_event-
>Advertising_Report[0].Address[2];
    SERVER_REMOTE_BDADDR[3] = le_advertising_event-
>Advertising_Report[0].Address[3];
    SERVER_REMOTE_BDADDR[4] = le_advertising_event-
>Advertising_Report[0].Address[4];
    SERVER_REMOTE_BDADDR[5] = le_advertising_event-
>Advertising_Report[0].Address[5];
   break;
```

- 検出されたあらゆる P2P サーバに対する接続を初期化します。

```
static void Connect_Request( void )
{
 tBleStatus result;
 APP_DBG_MSG("\r\n\r** CREATE CONNECTION TO SERVER **  \r\n\r");
 if (BleApplicationContext.Device_Connection_Status !=
APP_BLE_CONNECTED_CLIENT) {
  result = aci_gap_create_connection(
SCAN_P,
SCAN_L,
PUBLIC_ADDR, SERVER_REMOTE_BDADDR,
PUBLIC_ADDR,
CONN_P1
```

```
CONN_P2
0
SUPERV_TIMEOUT,
CONN_L1
CONN_L2);
```

- 接続が確立されたら「サービス・ディスカバリ手順」を開始します。

```
case EVT_LE_CONN_COMPLETE:
/**
* The connection is established
*/
connection_complete_event = (hci_le_connection_complete_event_rp0 *)
meta_evt->data;
BleApplicationContext.BleApplicationContext_legacy.connectionHandle =
connection_complete_event->Connection_Handle;
BleApplicationContext.Device_Connection_Status = APP_BLE_CONNECTED_CLIENT;

APP_DBG_MSG("\r\n\r** CONNECTION EVENT WITH SERVER \n");
handleNotification.P2P_Evt_Opcode = PEER_CONN_HANDLE_EVT;
handleNotification.ConnectionHandle =
 BleApplicationContext.BleApplicationContext_legacy.connectionHandle;
P2PC_APP_Notification(&handleNotification);
result = aci_gatt_disc_all_primary_services(
BleApplicationContext.BleApplicationContext_legacy.connectionHandle);
if (result == BLE_STATUS_SUCCESS) {
 APP_DBG_MSG("\r\n\r** GATT SERVICES & CHARACTERISTICS DISCOVERY  \n");
 APP_DBG_MSG("* GATT :  Start Searching Primary Services \r\n\r");
}
```

このステップにおいて、p2p_client_app.c の中で管理される GATT クライアント・イベント・ハンドラにすべての GATT イベントが転送されます。

P2P クライアント - アプリケーション制御 - GATT クライアント通信

p2p_client_app.c ファイルは以下の処理を行います。

- P2P クライアント・アプリケーションを初期化して、クライアント・イベント・ハンドラを登録します。
  - P2PC_APP_Init()
  - SVCCTL_RegisterCltHandler()
- ディスカバリ手順を開始して、リモート P2P サーバ・キャラクタリスティックを管理します。
  - aci_gatt_disc_all_char_of_service()
  - aci_gatt_disc_all_char_desc()
  - aci_gatt_write_char_desc()

```
case APP_BLE_DISCOVER_SERVICES:
APP_DBG_MSG("P2P_DISCOVER_SERVICES\n");
break;
```

```
case APP_BLE_DISCOVER_CHARACS:
APP_DBG_MSG("* GATT : Discover P2P Characteristics\n");
aci_gatt_disc_all_char_of_service(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PServiceHandle,
aP2PClientContext[index].P2PServiceEndHandle);
break;
case APP_BLE_DISCOVER_WRITE_DESC:
APP_DBG_MSG("* GATT : Discover Descriptor of TX - Write Characteritic\n");
aci_gatt_disc_all_char_desc(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PWriteToServerCharHdle,
aP2PClientContext[index].P2PWriteToServerCharHdle+2);
break;
case APP_BLE_DISCOVER_NOTIFICATION_CHAR_DESC:
APP_DBG_MSG("* GATT : Discover Descriptor of Rx - Notification
Characteritic\n");
aci_gatt_disc_all_char_desc(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PNotificationCharHdle,
aP2PClientContext[index].P2PNotificationCharHdle+2);
break;
case APP_BLE_ENABLE_NOTIFICATION_DESC:
APP_DBG_MSG("* GATT : Enable Server Notification\n");
aci_gatt_write_char_desc(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PNotificationDescHandle,
2,
(uint8_t *)&enable);
aP2PClientContext[index].state = APP_BLE_CONNECTED_CLIENT;
break;
case APP_BLE_DISABLE_NOTIFICATION_DESC :
APP_DBG_MSG("* GATT : Disable Server Notification\n");
aci_gatt_write_char_desc(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PNotificationDescHandle,
2,
(uint8_t *)&enable);
aP2PClientContext[index].state = APP_BLE_CONNECTED_CLIENT;
break;
```

- リモートデバイスのキャラクタリスティック・ハンドルを見つけて登録するための GATT イベントを管理します。
  - SVCCTL_EvtAckStatus_t Event_Handler()

```
uuid = UNPACK_2_BYTE_PARAMETER(&pr->Attribute_Data_List[idx]);
if(uuid == P2P_SERVICE_UUID){
APP_DBG_MSG("-- GATT : P2P_SERVICE_UUID FOUND - connection handle 0x%x \n",
aP2PClientContext[index].connHandle);
aP2PClientContext[index].P2PServiceHandle = UNPACK_2_BYTE_PARAMETER(&pr-
>Attribute_Data_List[idx-16]);
aP2PClientContext[index].P2PServiceEndHandle = UNPACK_2_BYTE_PARAMETER
(&pr->Attribute_Data_List[idx-14]);
```

```
aP2PClientContext[index].state = APP_BLE_DISCOVER_CHARACS ;
}
uuid = UNPACK_2_BYTE_PARAMETER(&pr->Handle_Value_Pair_Data[idx]);
/* store the characteristic handle not the attribute handle */
handle = UNPACK_2_BYTE_PARAMETER(&pr->Handle_Value_Pair_Data[idx-14]);
if(uuid == P2P_WRITE_CHAR_UUID){
APP_DBG_MSG("-- GATT : WRITE_UUID FOUND - connection handle 0x%x\n",
aP2PClientContext[index].connHandle);
aP2PClientContext[index].state = APP_BLE_DISCOVER_WRITE_DESC;
aP2PClientContext[index].P2PWriteToServerCharHdle = handle;
}
else if(uuid == P2P_NOTIFY_CHAR_UUID){
APP_DBG_MSG("-- GATT : NOTIFICATION_CHAR_UUID FOUND  - connection handle
0x%x\n", aP2PClientContext[index].connHandle);
aP2PClientContext[index].state = APP_BLE_DISCOVER_NOTIFICATION_CHAR_DESC;
aP2PClientContext[index].P2PNotificationCharHdle = handle;
}
```

P2P サーバ·サービスとキャラクタリスティック·ハンドルがディスカバーされたら、アプリケーションは次のことが可能となります。

- 「書込み」キャラクタリスティックを使用したリモートデバイスの制御

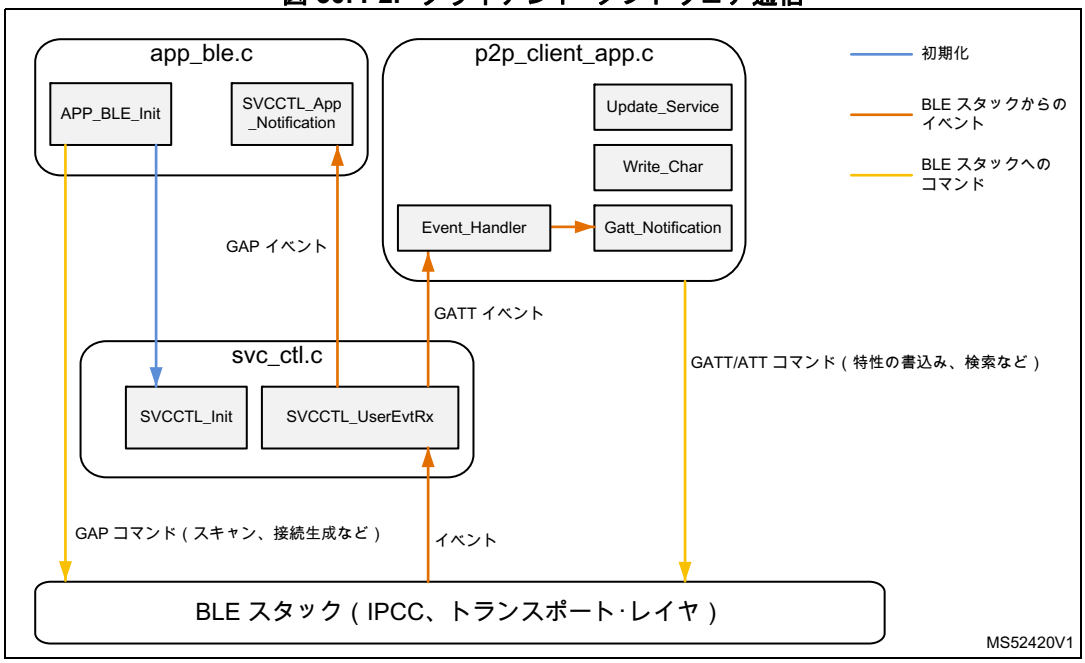
```
tBleStatus Write_Char(uint16_t UUID, uint8_t Service_Instance, uint8_t
*pPayload){
tBleStatus ret = BLE_STATUS_INVALID_PARAMS;
uint8_t index;
index = 0;
while((index < BLE_CFG_CLT_MAX_NBR_CB) && (aP2PClientContext[index].state
!= APP_BLE_IDLE)){
switch(UUID){
case P2P_WRITE_CHAR_UUID:
ret =aci_gatt_write_without_resp(aP2PClientContext[index].connHandle,
aP2PClientContext[index].P2PWriteToServerCharHdle,
2, /* charValueLen */
(uint8_t *)  pPayload);
break;
```

- 「通知」キャラクタリスティック経由の通知の受信

```
void Gatt_Notification(P2P_Client_App_Notification_evt_t *pNotification){
switch(pNotification->P2P_Client_Evt_Opcode){
case P2P_NOTIFICATION_INFO_RECEIVED_EVT: {
P2P_Client_App_Context.LedControl.Device_Led_Selection=pNotification-
>DataTransfered.pPayload[0];
switch(P2P_Client_App_Context.LedControl.Device_Led_Selection) {
case 0x01 : {
P2P_Client_App_Context.LedControl.Led1=pNotification-
>DataTransfered.pPayload[1];
if(P2P_Client_App_Context.LedControl.Led1==0x00){
BSP_LED_Off(LED_BLUE);
```

```
APP_DBG_MSG(" -- P2P CLIENT : NOTIFICATION RECEIVED - LED OFF \n\r");
} else {
BSP_LED_On(LED_BLUE);
APP_DBG_MSG(" -- P2P CLIENT : NOTIFICATION RECEIVED - LED ON\n\r");
}
break;
}
```

図 30. P2P クライアント·ソフトウェア通信



7.5 FUOTA アプリケーション

FUOTA は、Bluetooth LE サービスをインストールして新しい CPU2 ワイヤレス·スタック、CPU1 アプリケーション、または設定バイナリをダウンロード可能なスタンドアロン·アプリケーションです。

- （FUOTA アプリケーションが書き込まれる）アプリケーションの最初の6セクタの Flash メモリは絶対に削除されないことが必要となります。
- FUOTA アプリケーションによって次のことができるようになります。
  - CPU1 アプリケーション全体の更新
  - FUS によって適用される CPU2ワイヤレス·ファームウェアのダウンロード
  - CPU1ユーザ Flash メモリのあらゆるアドレスに対するユーザデータのダウンロード

7.5.1 CPU1ユーザ Flash メモリマッピング

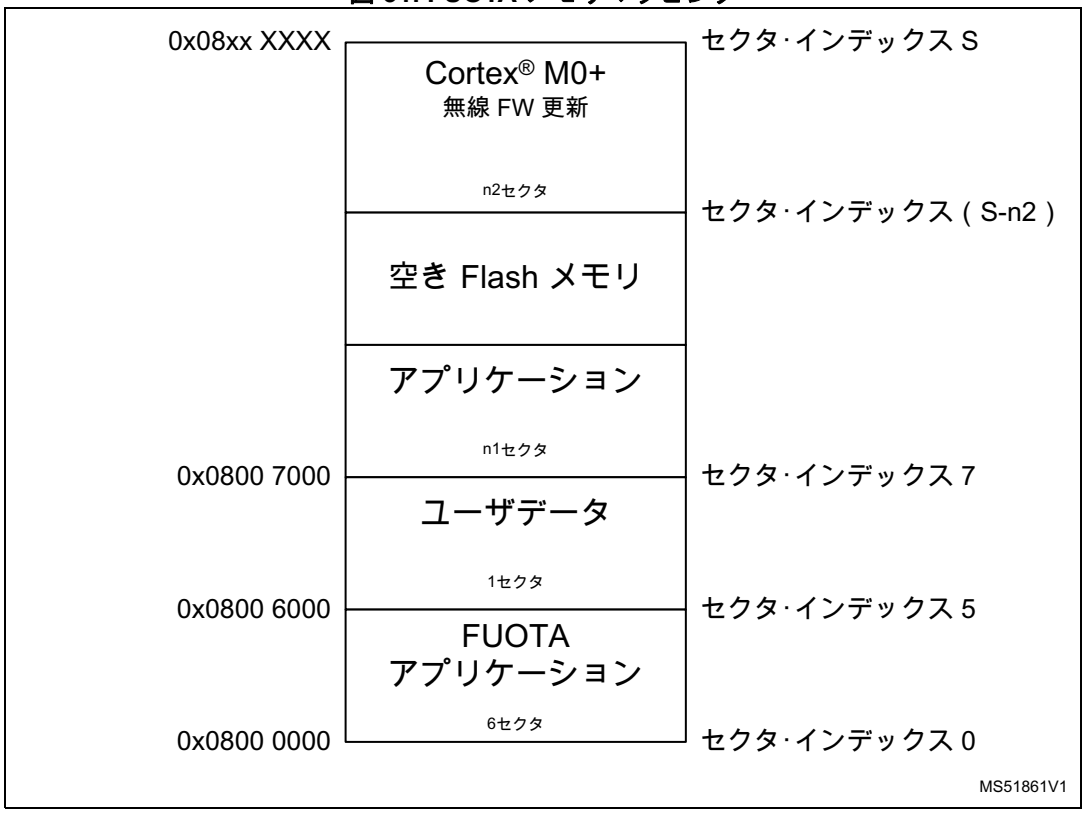
FUOTA Bluetooth LE アプリケーションは、自分自身の更新は行えませんが、次のことは可能です。

- 既存アプリケーションへのジャンプ（セクタ·インデックス 7）
- リモートデバイスの特定領域にデータをアップロードするための STMicroelectronics 独自 FUOTA GATT サービスとキャラクタリスティックの実行とインストール

ユーザデータセクションは、アプリケーション設定の一部を更新するために使用できます。

アプリケーション領域には、アプリケーション･スタンドアロン･バイナリが含まれています。この領域は、FUOTA アプリケーションで全体を更新できます。

図 31. FUOTA メモリマッピング



7.5.2 Bluetooth LE FUOTA アプリケーションの起動

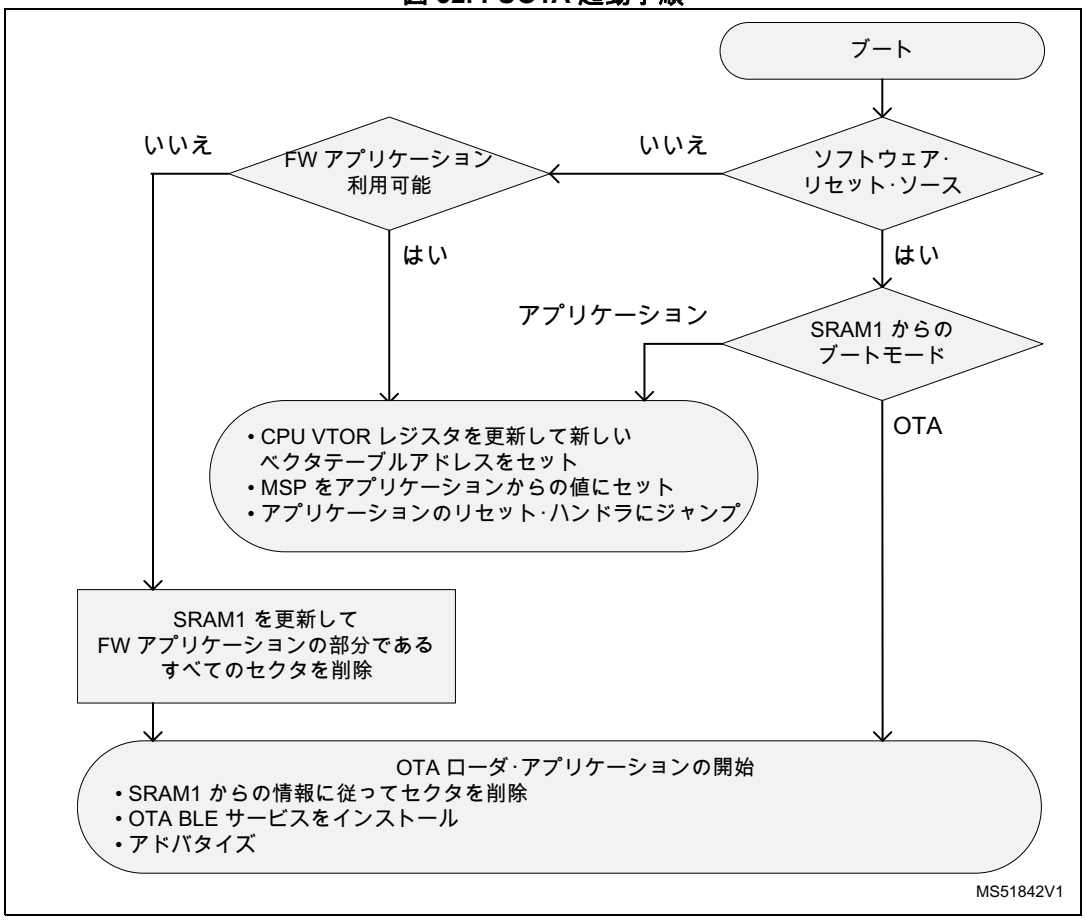
プロジェクト BLE_Ota のコンパイルとロードが行われた後に、アプリケーションは次のいずれかを実行できます。

- アプリケーション･セクタ（セクタ･インデックス 7）にバイナリコードが存在する場合の利用可能なアプリケーションへのジャンプ
  - BLE_Ota アプリケーションに関係した動作はそれ以上行いません。
- 以下をアドバタイジングするSTMicroelectronics 独自 FUOTA GATT サービスとキャラクタリスティックの開始
  - 「STM_OTA」であるローカル･ネーム･アドバタイジング･データ（AD）エレメント
  - デバイス IDが「STM32WB FW Update OTA application」である製造業者 AD エレメント

2番目の可能性では、リモートデバイスが新しいバイナリ（CPU2 ワイヤレススタック、CPU1 アプリケーション、またはユーザデータ･ファームウェア更新）を更新できます。

注： **STMicroelectronics 独自 FUOTA GATT サービスとキャラクタリスティックのみが使用される場合には、（7以降から）アプリケーションセクタを消去することが重要となります。**

図 32. FUOTA 起動手順



7.5.3 Bluetooth LE FUOTA サービスとキャラクタリスティックの仕様

Bluetooth LE FUOTA アプリケーション（BLE_Ota プロジェクト）は、次のキャラクタリスティックを備える GATT サービスとしてエクスポートされます。

- 新しいバイナリを格納する場所の情報を提供するベースアドレス
- 新しいバイナリファイルのアップロード後にアプリケーションのリブートを確認するファイル·アップロード·リブート確認
- データ（パケットに分割されたバイナリファイル）を転送する OTA 元データ

表 23. FUOTA サービスとキャラクタリスティック UUID

グループ	サービス	特徴	サイズ	モード	UUID
LED ボタン制御	OTA FW 更新	-	-	-	0000FE20-cc7a-482a-984a-7f2ed5b3e58f
	-	ベースアドレス	4 バイト	書込み	0000FE22-8e22-4541-9d4c-21edae82ed19
	-	ファイル·アップロード·リブート確認	1 バイト	通告	0000FE23-8e22-4541-9d4c-21edae82ed19
	-	OTA 元データ	20 バイト	応答なし書込み	0000FE24-8e22-4541-9d4c-21edae82ed19

表 24. ベースアドレス･キャラクタリスティックの仕様

ビット LSB	[0:7]	[8:31]
名前	アクション	アドレス
値	– 0x00 : 全アップロードの停止 – 0x01 : M0+ ファイル･アップロードの開始 – 0x02 : M0+ ファイル･アップロードの開始 – 0x07 : ファイル･アップロードの完了 – 0x08 : アップロードのキャンセル	0x007000

表 25. ファイル･アップロード確認リブート･リクエスト･キャラクタリスティックの仕様

オクテット LSB	0
名前	通告
値	0x01 リブート

表 26. 元データ･キャラクタリスティックの仕様

オクテット LSB	0	1	.....	19
名前	元のデータ			
値	ファイル･データ			

7.5.4 新しい CPU1 アプリケーション･バイナリをアップロードするフローの記述例

新しいバイナリをアップロードする手順は 2種類あります。

- STMicroelectronics 独自 FUOTA GATT サービスとキャラクタリスティック･アプリケーションのみをロードします。(セクタ 7にはアプリケーション･バイナリが存在しません)
- リブート･リクエスト･キャラクタリスティックのサポートにより、アプリケーションはすでに動作中です。

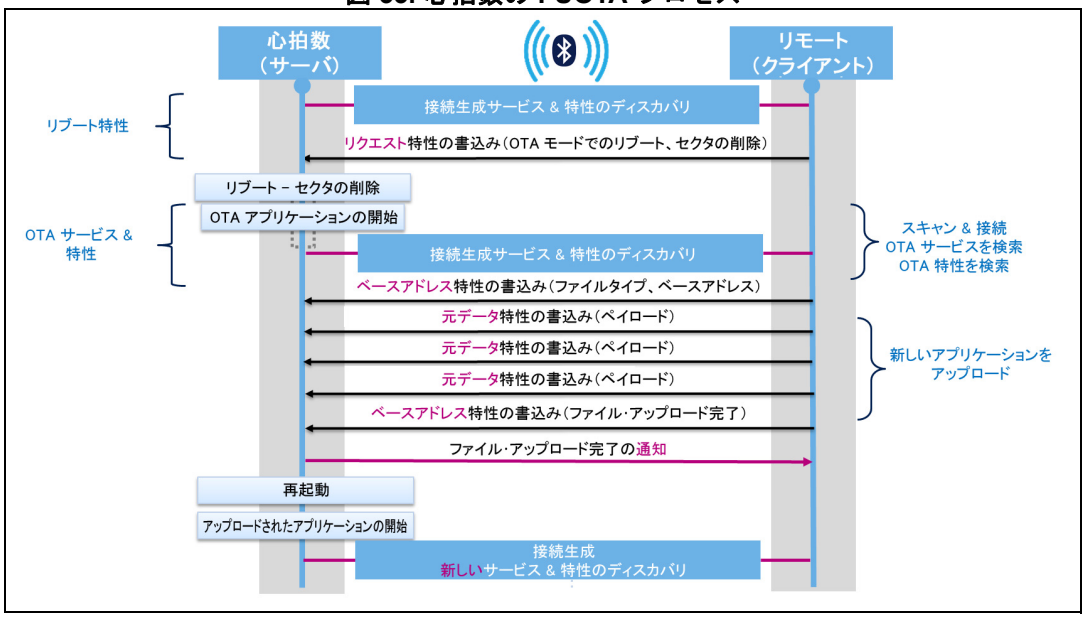
表 27. リブート･リクエスト･キャラクタリスティックの仕様

オクテット LSB	0	1	2
名前	ブートモード	セクタ･インデックス	消去セクタ数
値	– 0x00 アプリケーション – 0x01 FUOTA アプリケーション	07 → 0x0800 7000	0x00 ... 0xFF

リブート･リクエスト･キャラクタリスティックを含むアプリケーションから始まるこれは、CPU1アプリケーションを更新するフローです。

1. Bluetooth LE アプリケーションには、リブート･キャラクタリスティックが含まれています。
2. GAP 接続が確立されると、リモート GATT クライアント･デバイスはサービスとキャラクタリスティックを再検索します（リブート･リクエスト･キャラクタリスティックが検出されます）。
3. 次に、FUOTA アプリケーションに切り替えるため、リモートデバイスは、ブートモード･オプションと消去するセクタの情報とともにリブート･リクエスト･キャラクタリスティックを書き込みます。
4. STMicroelectronics 独自仕様 FUOTA GATT サービスとキャラクタリスティック･アプリケーションでリブートするために、この段階で Bluetooth LE リンクが切断されます。
5. リブート･キャラクタリスティックから提供される情報を用いてアプリケーション･セクタが消去され、STMicroelectronics 独自仕様 FUOTA GATT サービスとキャラクタリスティック･アプリケーションがアドバタイジングを開始します。
6. FUOTA サービスとキャラクタリスティックをディスカバーするために、リモートデバイスによって新しい接続が確立される必要があります。
7. ベースアドレス･キャラクタリスティックは、新しいバイナリのアップロードを開始するために使用されます。
8. すべてのデータは元データ･キャラクタリスティックを経由して転送され、受信後に直接 Flash メモリにプログラムされます。
9. ベースアドレス･キャラクタリスティックによってファイル転送の終了が確認されます。
10. 受信ファイルの確認は、ファイル･アップロード確認キャラクタリスティックによって示されます。
11. この段階で、FUOTA アプリケーションは新しいバイナリの整合性を確認し、ダウンロード済みの新しいアプリケーションを起動するためにリブートします。
12. アプリケーションの整合性が確実ではない場合、FUOTA アプリケーションをリブートするためにアプリケーション･セクタが消去されます。

図 33. 心拍数の FUOTA プロセス



7.5.5 スマートフォンのアプリケーション例

プロジェクトにはリブート･リクエスト･キャラクタリスティックが実装されています。

- BLE_HeartRate_ota
- BLE_P2pServer_ota

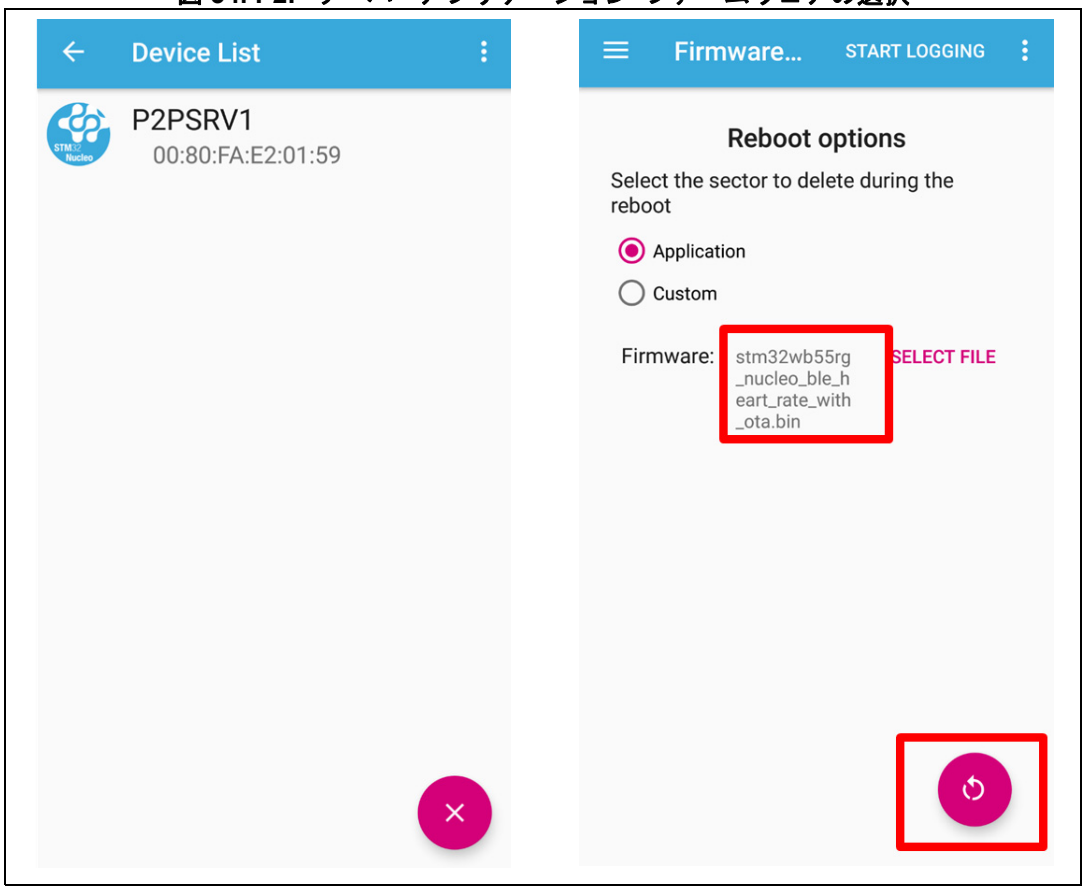
どちらのプロジェクトも、そのアドバタイジング･エレメントの中に OTA リブート･リクエスト･ビットマスクが含まれています。これは、リモート（スキャナ）が、リブート･リクエスト･キャラクタリスティックの存在に関する情報を迅速に取得する方法です。

ST Bluetooth LE センサ･モバイル･アプリケーションは、このリブート･リクエスト･キャラクタリスティックの検出をサポートしています。

たとえば、P2P サーバ･アプリケーションから心拍数アプリケーションへの移行などです。

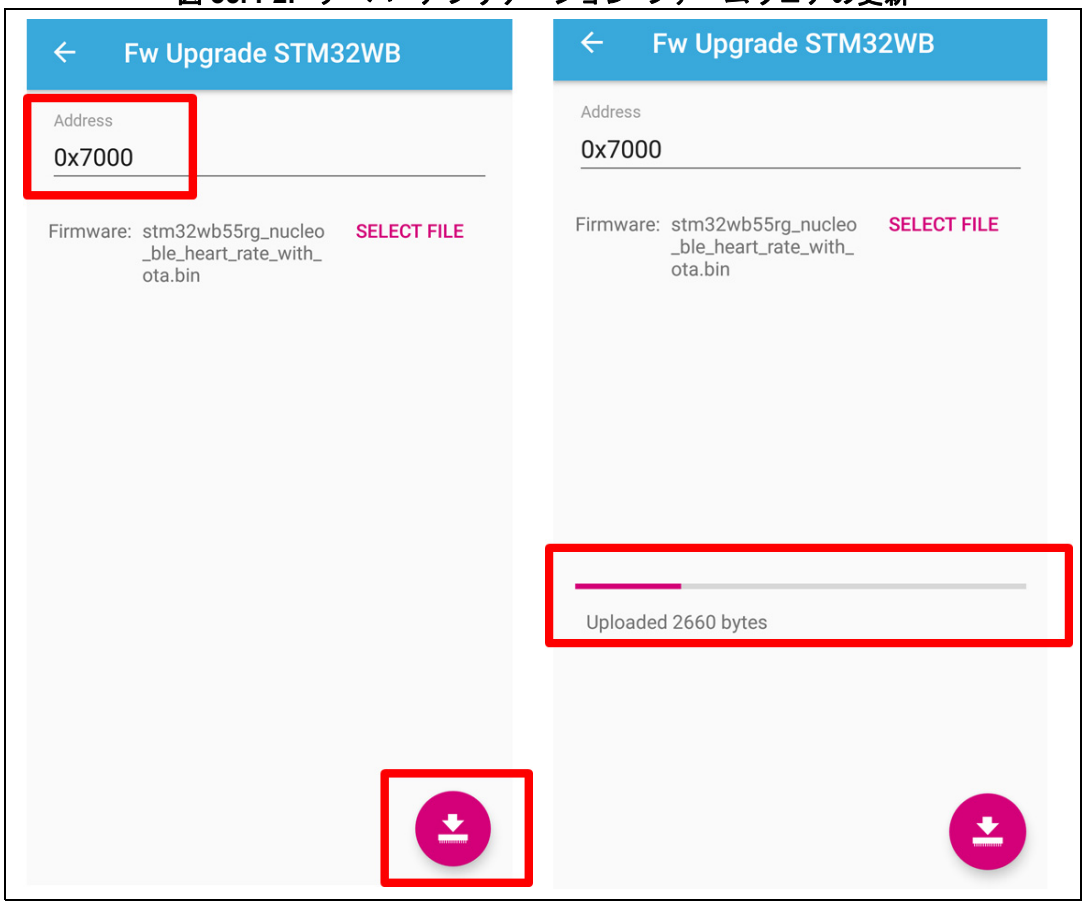
- BLE_Ota プロジェクトをコンパイルして、アドレス 0x0800 0000にロードします。
- BLE_p2pServer_ota プロジェクトをコンパイルして、アドレス 0x0800 7000にロードします。
- デバイスをリブートします。
  - この段階で、P2P サーバはその存在をアドバタイズします。
- ST Bluetooth LE センサ･モバイル･アプリケーションで P2P サーバをディスカバーして接続します。
- リブート･パネルに移動します。
- バイナリ「BLE_HeartRate_ota」（デモンストレーション前にスマートフォンのメモリにコピーします）を選択します。
- アップロードをクリックします。
  - この段階で、リブート･リクエスト･キャラクタリスティックを使用して、消去するセクタと次のリブート･フェーズ（FUOTA アプリケーション）に関する情報を提供します。

図 34. P2P サーバ - アプリケーション·ファームウェアの選択



リブートされると、アプリケーション·バイナリ·ファイルをアップロードするアドレスが選択されています。デフォルト·アドレスは 0x7000（セクタ 7 - アプリケーション）です。この段階でも、必要に応じてアップロードするバイナリファイルを変更することができます。

図 35. P2P サーバ - アプリケーション·ファームウェアの更新



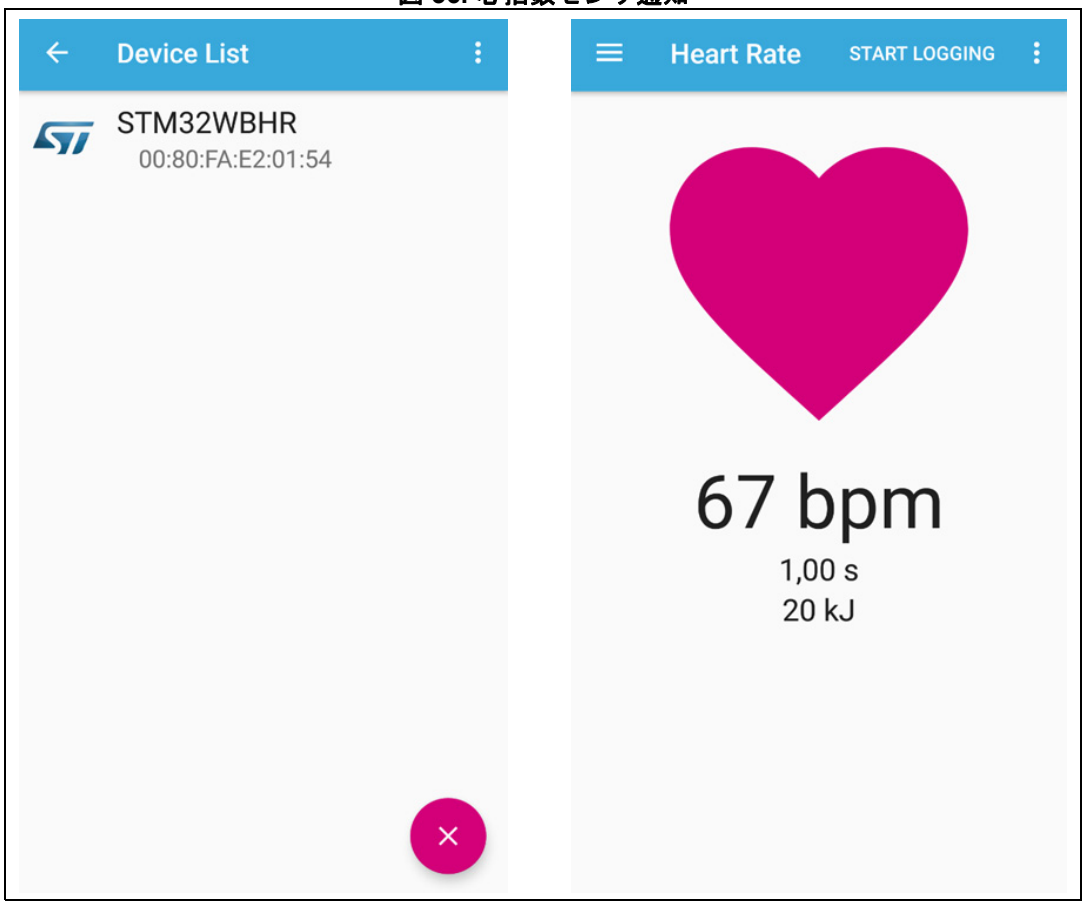
アップロードが完了すると、リブート手順が実行されて新しいアプリケーションが起動します。

次に、新しいスキャン手順を実行し、心拍数センサ·アドバタイジング·パケットをディスカバーして接続します。

デバイスに接続した後、心拍数測定値がセンサから通知されます。

注： **スマートフォン·アプリケーションによって、GATT データベースがリモート Bluetooth アドレスに関連付けられます。この問題を解決するために、FUOTA アプリケーション·アドバタイジング·アドレスが 1だけ増やされます。**

図 36. 心拍数センサ通知



7.5.6 リブート・リクエスト・キャラクタリスティックの使用方法

どのアプリケーションが使用されていても、リブート・リクエスト・キャラクタリスティックをサービスに統合して、アプリケーションを FUOTA アプリケーション・モードでリブートすることができます。

アプリケーションは、「BLE_HeartRate_ota」と「BLE_p2pServer_ota」の例を用いてアドレス 0x08000700 にロードする必要があります。設定は次のように行われます。

- ble_conf.h：OTA リブート・キャラクタリスティックを定義

```
/******************************************************************************
 * Over The Air Feature (OTA) - STM Proprietary

*******************************************************************************/
#define BLE_CFG_OTA_REBOOT_CHAR           1 /**< REBOOT OTA MODE CHARACTERISTIC */
```

- app_ble.c

```
/**
   * Initialization of ADV - Ad Manufacturer Element - Support OTA Bit Mask
 */
#if(BLE_CFG_OTA_REBOOT_CHAR != 0)
```

```
  manuf_data[sizeof(manuf_data)-8] = CFG_FEATURE_OTA_REBOOT;
#endif
```

- p2p_stm.c：キャラクタリスティックを追加（ミドルウェア）

```
#if(BLE_CFG_OTA_REBOOT_CHAR != 0)
    /**
     *  Add Boot Request Characteristic
     */
    aci_gatt_add_char(aPeerToPeerContext.PeerToPeerSvcHdle,
                      BM_UUID_LENGTH,
                      (Char_UUID_t *)BM_REQ_CHAR_UUID,
                      BM_REQ_CHAR_SIZE,
                      CHAR_PROP_WRITE_WITHOUT_RESP,
                      ATTR_PERMISSION_NONE,
                      GATT_NOTIFY_ATTRIBUTE_WRITE,
                      10
                      0,
                      &(aPeerToPeerContext.RebootReqCharHdle));
#endif
```

- p2p_stm.c：GATT レベルでリクエストを受信してアプリケーションに通知（ミドルウェア）

```
else if(attribute_modified->Attr_Handle ==
(aPeerToPeerContext.RebootReqCharHdle + 1))
{
BLE_DBG_P2P_STM_MSG("-- GATT : REBOOT REQUEST RECEIVED\n");
Notification.P2P_Evt_Opcode = P2PS_STM_BOOT_REQUEST_EVT;
Notification.DataTransfered.Length=attribute_modified->Attr_Data_Length;
Notification.DataTransfered.pPayload=attribute_modified->Attr_Data;
P2PS_STM_App_Notification(&Notification);
```

- p2p_server_app.c：リブート･リクエストを管理（アプリケーション）

```
void P2PS_STM_App_Notification(P2PS_STM_App_Notification_evt_t
*pNotification)
{
  switch(pNotification->P2P_Evt_Opcode)
  {
#if(BLE_CFG_OTA_REBOOT_CHAR != 0)

    case P2PS_STM_BOOT_REQUEST_EVT:
      APP_DBG_MSG("-- P2P APPLICATION SERVER : BOOT REQUESTED\n");
      APP_DBG_MSG(" \n\r");

      *(uint32_t*)SRAM1_BASE = *(uint32_t*)pNotification-
>DataTransfered.pPayload;
      NVIC_SystemReset();
      break;
#endif
```

参考資料

7.5.7 CPU1 アプリケーションの電源喪失リカバリ·メカニズム

BLE_ota アプリケーションは、CPU1 アプリケーション更新中の電源喪失リカバリ·メカニズムを備えています。

CPU1 アプリケーション·ファームウェア更新中の電源喪失の管理に使用される 2つのタグは次のものです。

1. MagicKeywordAddress : ロードするバイナリイメージの先頭から 0x140 の位置にマッピングされる必要があります。
2. MagicKeywordvalue : MagicKeywordAddress にある BLE_ota アプリケーションによってチェックされます。

新しいアプリケーションの Flash 書込み中にリンクがドロップした場合、BLE_ota アプリケーションが喪失を検出し、プログラムされたセクタを自動的に消去します。このメカニズムによって、間違ったアプリケーションのリブートが防止されます。

```
/**
 * These are the two tags used to manage a power failure during CM4 Application OTA FW Update
 * The MagicKeywordAdress shall be mapped @0x140 from start of the binary image
 * The MagicKeywordvalue is checked in the ble_ota application
 */
PLACE_IN_SECTION("TAG_OTA_END") const uint32_t MagicKeywordValue = 0x94448A29 ;
PLACE_IN_SECTION("TAG_OTA_START") const uint32_t MagicKeywordAddress = (uint32_t)&MagicKeywordValue;
```

```
define region OTA_TAG_region   = mem:[from (__ICFEDIT_region_ROM_start__ + 0x140)   to (__ICFEDIT_region_ROM_start__ + 0x140 + 4)];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy { readwrite };
do not initialize  { section .noinit,
                     section MAPPING_TABLE,
                                        section MB_MEM1 };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

keep { section TAG_OTA_START};
keep { section TAG_OTA_END };
place in OTA_TAG_region   { section TAG_OTA_START };
place in ROM_region   { readonly, last section TAG_OTA_END };
```

7.6 アプリケーションのヒント

7.6.1 Bluetooth デバイスアドレスの設定方法

すべてのBluetooth デバイスにはそれらを一意に識別するアドレスが必要です。

STM32WB デバイスでは、以下のアドレスタイプがサポートされています。

- パブリックアドレス
- ランダムアドレス（スタティックアドレスとプライベートアドレスのいずれか）

デバイスアドレスはパブリックとランダムのいずれかとなります。デバイスアドレスは、パブリックとランダムのいずれも 48ビット長であり、コロン区切りの 16進値（AA:BB:CC:DD:EE:FF など）で表記されます。

パブリックデバイスアドレスは、IEEE 登録局から取得した有効な組織固有識別子（OUI）を使用して、IEEE 802-2001 規格に準拠して生成される必要があります。パブリックデバイスアドレスは、MAC アドレスと呼ばれています。

Bluetooth LE デバイスがランダムアドレスをどのように生成可能であるかの詳細については、Core Specification v5.0 を参照してください。

STM32WB は 64ビット固有デバイス識別を提供します。

- 24ビットの企業 ID（STMicroelectronics は 0x00 80 E1）
- 8ビットのデバイス ID（STM32WB は 0x05）
- 個々のデバイスを識別するための 32ビットの固有デバイス番号

アプリケーションが異なるパブリックアドレスを使用する必要がある場合、適切な組織からアドレスを取得して、最終製品の永続的メモリ位置（マイクロコントローラの Flash メモリあるいは OTP、または外部ストレージ領域）に格納する必要があります。

STM32WB の初期化フェーズの間に、アプリケーションはこのアドレスを設定する必要があります。

このパブリックアドレスをセットする ACI コマンドは次のようになります。

```
tBleStatus aci_hal_write_config_data(uint8_t offset, uint8_t len, const
uint8_t *val).
```

パラメータは以下のようにセットしてください。

- オフセット：0x00
- 長さ：0x06
- 値：パブリックアドレス値に対するポインタ - 0xaabbccddeeff（6バイト配列）など

コマンド aci_hal_write_config_data では Flash メモリの中のデータの体系的保存は行われませんので、アプリケーション･マイクロプロセッサは、あらゆる Bluetooth LE 動作を開始する前ならびに毎回の起動またはリセットの後に、ワイヤレス･マイクロプロセッサにコマンド aci_hal_write_config_data を送信する必要があります。

次の疑似コードは、アプリケーションから MAC アドレスを設定する方法の例です。

```
uint8_t bdaddr[] = {0xFF, 0xEE, 0xDD, 0xCC, 0xBB, 0xAA};
ret=aci_hal_write_config_data(0x00, 0x06, bdaddr);
if(ret) { PRINTF("Setting address failed.\n")}
```

Bluetooth LE デバイスは、ランダムアドレスを使用することもできます。アドレス値は、tBleStatus aci_hal_read_config_data(uint8_t offset, uint16_t data_len, uint8_t *data_len_out_p, uint8_t *data) を用いてアプリケーションから読み出すことができます。コマンドのパラメータ･オフセットは 0x80 にセットします。

あるいは、アプリケーションは、毎回のリセット後に int hci_le_set_random_address(tBDAddr bdaddr) コマンドを使用して外部ホスト･プロセッサからランダムアドレスをセットすることもできます。ランダムアドレスが hci_le_set_random_address コマンドを通じてセットされない場合、アドレス生成は、上記のスタックによって独立に処理されます。

STM32WB デバイスの 64ビット UID を使用して固有の Bluetooth LE 48ビット･デバイスアドレスを導出することができます。また、OTP レジスタから Bluetooth LE 48ビット･デバイスアドレスを取得することもできます。

```
const uint8_t* BleGetBdAddress( void ) {
 uint8_t *otp_addr;
 const uint8_t *bd_addr;
 uint32_t udn;
 uint32_t company_id;
 uint32_t device_id;
 udn = LL_FLASH_GetUDN();
```

```
 if(udn != 0xFFFFFFFF) {
  company_id = LL_FLASH_GetSTCompanyID();
  device_id = LL_FLASH_GetDeviceID();

    bd_addr_udn[0] = (uint8_t)(udn & 0x000000FF);
    bd_addr_udn[1] = (uint8_t)( (udn & 0x0000FF00) >> 8 );
    bd_addr_udn[2] = (uint8_t)( (udn & 0x00FF0000) >> 16 );
    bd_addr_udn[3] = (uint8_t)device_id;
    bd_addr_udn[4] = (uint8_t)(company_id & 0x000000FF);;
    bd_addr_udn[5] = (uint8_t)( (company_id & 0x0000FF00) >> 8 );

    bd_addr = (const uint8_t *)bd_addr_udn;
  }
  else {
   otp_addr = OTP_Read(0);
    if(otp_addr) {
     bd_addr = ((OTP_ID0_t*)otp_addr)->bd_address;
     }
     else {
      bd_addr = M_bd_addr;
    }
  }
 return bd_addr;
}
```

7.6.2 シーケンサにタスクを追加する方法

- タスク ID を宣言 - app_conf.h -

```
/**< Add in that list all tasks that may send a ACI/HCI command */
typedef enum
{
    CFG_TASK_ADV_CANCEL_ID,
    CFG_TASK_SW1_BUTTON_PUSHED_ID
    CFG_TASK_HCI_ASYNCH_EVT_ID,
    CFG_LAST_TASK_ID_WITH_HCICMD,  /**< Shall be LAST in the list */
} CFG_Task_Id_With_HCI_Cmd_t;

/**< Add in that list all tasks that never send a ACI/HCI command */
typedef enum
{
   CFG_FIRST_TASK_ID_WITH_NO_HCICMD = CFG_LAST_TASK_ID_WITH_HCICMD - 1,
/**< Shall be FIRST in the list */
    CFG_TASK_SYSTEM_HCI_ASYNCH_EVT_ID,
   CFG_LAST_TASK_ID_WITHO_NO_HCICMD
/**< Shall be LAST in the list */
} CFG_Task_Id_With_NO_HCI_Cmd_t;
#define UTIL_SEQ_CONF_TASK_NBR    CFG_LAST_TASK_ID_WITHO_NO_HCICMD
```

- コールバック関数をタスクに登録 - “アドバタイジングのキャンセル” - app_ble.c

```
SCH_RegTask(CFG_TASK_ADV_CANCEL_ID, Adv_Cancel);
```

- 優先順位の高いタスクを開始 - app_ble.c

```
SCH_SetTask(1 << CFG_TASK_ADV_CANCEL_ID, CFG_SCH_PRIO_0);
```

7.6.3 タイマ･サーバの使用方法

- コールバック関数付きでタイマを生成

```
/**
   * Create timer to handle the Led Switch OFF
   */
HW_TS_Create(CFG_TIM_PROC_ID_ISR,
&(BleApplicationContext.SwitchOffGPIO_timer_Id), hw_ts_SingleShot,
Switch_OFF_GPIO);
```

- タイムアウト付きタイマを開始します。

```
HW_TS_Start(BleApplicationContext.SwitchOffGPIO_timer_Id,
(uint32_t)LED_ON_TIMEOUT);
```

- タイマを停止します。

```
HW_TS_Stop(BleApplicationContext.SwitchOffGPIO_timer_Id);
```

- コールバック関数の例

```
static void Switch_OFF_GPIO(){
 BSP_LED_Off(LED_GREEN);
}
```

7.6.4 Bluetooth LE スタックの開始方法 - SHCI_C2_BLE_Init()

```
SHCI_C2_Ble_Init_Cmd_Packet_t ble_init_cmd_packet =
  {
    0,0,0,                                  /**< Header unused */
    0,                                      /** pBleBufferAddress not used */
    0,                                      /** BleBufferSize not used */
    CFG_BLE_NUM_GATT_ATTRIBUTES,
    CFG_BLE_NUM_GATT_SERVICES,
    CFG_BLE_ATT_VALUE_ARRAY_SIZE,
    CFG_BLE_NUM_LINK,
    CFG_BLE_DATA_LENGTH_EXTENSION,
    CFG_BLE_PREPARE_WRITE_LIST_SIZE,
    CFG_BLE_MBLOCK_COUNT,
    CFG_BLE_MAX_ATT_MTU,
    CFG_BLE_SLAVE_SCA,
    CFG_BLE_MASTER_SCA,
    CFG_BLE_LSE_SOURCE,
    CFG_BLE_MAX_CONN_EVENT_LENGTH,
    CFG_BLE_HSE_STARTUP_TIME,
    CFG_BLE_VITERBI_MODE,
    CFG_BLE_LL_ONLY,
```

```
    0                                   /** TODO Should be read from HW */
  };
```

CFG_BLE_NUM_GATT_ATTRIBUTES

特定の Bluetooth LE ユーザアプリケーションに対して、GATT データベースに格納可能なすべての必要なキャラクタリスティック（サービスを除く）に関連する属性レコードの最大数です。

それぞれのキャラクタリスティックに対して、属性レコードの数はキャラクタリスティック·プロパティに応じて次のように2から5の間となります。

- 最低で2つ（1つは宣言用、1つは値用）
- 追加プロパティ（通知または通告、ブロードキャスト、拡張プロパティ）ごとに 1レコードを追加

標準属性プロファイルと GAP サービス·キャラクタリスティックに関連したレコードのために、合計の計算値に9を加える必要があり、GAP レイヤと GATT レイヤの初期化時には自動的に加算されます。

- 最小値：<ユーザ属性数> + 9
- 最大値：ユーザアプリケーションによって定義される GATT データベースによる

CFG_BLE_NUM_GATT_SERVICES

GATT データベースに格納可能なサービスの最大数を定義します。GAP と GATT サービスは初期化時に自動的に追加されますので、このパラメータは2だけ増やしたユーザ·サービス数である必要があることに注意してください。

- 最小値：<ユーザ·サービス数> + 2
- 最大値：ユーザアプリケーションによって定義される GATT データベースによる

CFG_BLE_ATT_VALUE_ARRAY_SIZE

属性値のためのストレージ領域のサイズです。

それぞれのキャラクタリスティックは attrValueArrSize 値に次のように影響します。

- キャラクタリスティック値の長さに次の値を加算します。
  - 5バイト：キャラクタリスティック UUID が 16ビットである場合
  - 19バイト：キャラクタリスティック UUID が 128ビットである場合
  - 2バイト：キャラクタリスティックにサーバ設定ディスクリプタが含まれている場合
  - 2バイト * CFG_BLE_NUM_LINK: キャラクタリスティックにクライアント設定ディスクリプタが含まれている場合
  - 2バイト：キャラクタリスティックに拡張プロパティが含まれている場合

それぞれのディスクリプタは attrValueArrSize 値に次のように影響します。

- ディスクリプタ長

CFG_BLE_NUM_LINK

サポートされている Bluetooth LE リンクの最大数です。

- 最小値：1
- 最大値: 8

CFG_BLE_DATA_LENGTH_EXTENSION

拡張パケット長 Bluetooth LE 5.0 機能を無効／有効にします。

- 無効：0
- 有効：1

CFG_BLE_PREPARE_WRITE_LIST_SIZE

サポートされている「Prepare Write Request」の最大数です。最小必要数は、次の DEFAULT_PREP_WRITE_LIST_SIZE マクロを使用して計算できます。

```
#define DIV_CEIL(x, y)        (((x) + (y) - 1) / (y))

/**
 * DEFAULT_ATT_MTU: minimum mtu value that GATT must support.
 * 5.2.1 ATT_MTU, BLUETOOTH SPECIFICATION Version 4.2 [Vol 3, Part G]
 */
#define DEFAULT_ATT_MTU                  (23)
/**
 * DEFAULT_MAX_ATT_SIZE: maximum attribute size.
 */
define DEFAULT_MAX_ATT_SIZE             (512)

/**
 * PREP_WRITE_X_ATT(max_att): compute how many Prepare Write Request are
needed
 *
 *
 */
#define PREP_WRITE_X_ATT(max_att)       (DIV_CEIL(max_att, DEFAULT_ATT_MTU
- 5U) * 2)
/**
 * DEFAULT_PREP_WRITE_LIST_SIZE: default minimum Prepare Write List size.
 */
#define DEFAULT_PREP_WRITE_LIST_SIZE
PREP_WRITE_X_ATT(DEFAULT_MAX_ATT_SIZE)
```

- 最小値：上記マクロ参照
- 最大値：最小必要数を上回る値を指定できますが、推奨されません

CFG_BLE_MBLOCK_COUNT

Bluetooth LE スタックに割り当てられるメモリブロック数です。最小必要数は、次の MBLOCKS_CALC マクロを使用して計算できます。

```
#define MEM_BLOCK_SIZE                  (32)
/**
 * MEM_BLOCK_X_MTU (mtu): compute how many memory blocks are needed to
compose an ATT
 * Packet with ATT_MTU = mtu.
```

```
 * 7.2 FRAGMENTATION AND RECOMBINATION, BLUETOOTH SPECIFICATION Version 4.2
 * [Vol 3, Part A]
 */
#define MEM_BLOCK_X_TX (mtu)              (DIV_CEIL((mtu) + 4U,
MEM_BLOCK_SIZE) + 1U)
#define MEM_BLOCK_X_RX (mtu, n_link)      ((DIV_CEIL((mtu) + 4U,
MEM_BLOCK_SIZE) + 2U) * (n_link) + 1)
#define MEM_BLOCK_X_MTU (mtu, n_link)     (MEM_BLOCK_X_TX(mtu) +
MEM_BLOCK_X_RX(mtu, (n_link)))

/**
 * Minimum number of blocks required for secure connections
 */
#define MBLOCKS_SECURE_CONNECTIONS      (4)

/**
 * MBLOCKS_CALC(pw, mtu, n_link): minimum number of buffers needed by the
stack.
 * This is the minimum racomanded value and depends on:
 *  - pw: size of Prepare Write List
 *  - mtu: ATT_MTU size
 *  - n_link: maximum number of simultaneous connections
 */
#define MBLOCKS_CALC(pw, mtu, n_link)    ((pw) + MAX(MEM_BLOCK_X_MTU(mtu,
n_link), (MBLOCKS_SECURE_CONNECTIONS)))
```

- 最小値：上記マクロ参照
- 最大値：値を増やすとデータ･スループット性能が向上しますが、メモリ使用量が増えます。

CFG_BLE_MAX_ATT_MTU

サポートされている最大 ATT MTU サイズです。

- 最小値：23
- 最大値: 512

CFG_BLE_SLAVE_SCA

Bluetooth LE 接続されたスレーブモードにおいて（CONNECT_REQ PDU でマスタから送信されるスリープクロック精度との組み合わせで）ウィンドウ拡張の計算に必要なスリーブ･クロック精度（ppm 値）については、Bluetooth LE 5.0 Specifications - Vol 6 - Part B - Chapter 4.5.7および 4.2.2を参照してください。

- 最小値：0
- 最大値: 500（仕様によって認められている最悪値）

CFG_BLE_MASTER_SCA

マスタモードで処理されるスリープクロック精度です。接続とアドバタイジング·イベントのタイミングの決定に使用されます。スレーブがウィンドウ拡張の計算に使用する CONNEC_REQ PDU の中でスレーブに転送されます（CFG_BLE_SLAVE_SCAならびに[7], v5.0 Vol 6 - Part B - Chapter 4.5.7 および 4.2.2参照）。

可能な値：

- 251～500ppm: 0
- 151～250ppm：1
- 101～150ppm：2
- 76～100ppm：3
- 51～75ppm：4
- 31～50ppm：5
- 21～30ppm：6
- 0～20ppm：7

CFG_BLE_LSE_SOURCE

32kHz 低速クロックのソースです。

- 外部クリスタル LSE：0 - 較正なし
- 内部 RO（LSI）：1 - 外部条件（温度）によりこのオシレータの精度は変化するため、毎秒較正を行ってタイミングの影響を受けやすい Bluetooth LE 操作が正しく動作することを保証します。

CFG_BLE_MAX_CONN_EVENT_LENGTH

このパラメータは、スレーブ接続イベントの最長時間を決定します。この時間に達すると、（HCI_CREATE_CONNECTION HCI コマンドでマスタによって指定された CE_length パラメータにはよらず）スレーブは現在の接続イベントをクローズします（625/256µs（～2.44µs）単位）。

- 最小値：0（0が指定された場合、マスタとスレーブは TX-RX 交換を接続イベントあたり 1 回だけ行います）
- 最大値: 1638400（4000ms）。これ以上の値（最大 0xFFFFFFFF）を指定することも可能ですが、最大接続時間は指定どおりに 4000ms となります。この場合にはパラメータは適用されず、スレーブ側で計算された予想 CE 長は短縮されません。

CFG_BLE_HSE_STARTUP_TIME

ハイスピード（16 または 32MHz）クリスタルオシレータの起動時間（625/256µs（～2.44µs）単位）です。

- 最小値：0
- 最大値: 820（～2ms）。これ以上の値を指定することも可能ですが、スタックで実行される値は強制的に ～2ms となります。

CFG_BLE_VITERBI_MODE

Bluetooth LE LL 受信の Viterbi 実装です。

- 0：有効化
- 1：無効化

CFG_BLE_LL_ONLY

Bluetooth LE リンク･レイヤのみモードを有効化／無効化します。このモードでは、ホストスタック（GAP/GATT/SMP など）コードは存在しません。リンク･レイヤのみが実装され、HCI コマンド経由で処理できます。

- 「LL のみ」モードを無効化：0
- 「LL のみ」モードを有効化：1

7.6.5 データ･スループットを最大化する方法

GATT サーバ通知が次のリンク･レイヤ･パラメータで使用される場合に、最大のデータ･スループットが得られます。

- 接続間隔：50 ms
- Min_CE_Length および Max_CE_Lenght：0x50 (50ms)

接続が確立されると、マスタデバイスは、MAX_ATT_MTU 値を取得するために、aci_gatt_exchange_configを送信します。

データ交換は、最大通知長に対応した（MAX_ATT_MTU – 3）に制限されます。

- サポートされている場合には、リンクを 2M にセットします。

最大 PDU_length = 247（251 - 4）で LE データのフラグメンテーションを防止するには：

- hci_le_set_data_length コマンド hci_le_set_data_length(conn_handle, 251, 2120) を使用します。

フラグメンテーションを防止するには：

- MAX_ATT_MTU = 250 かつ le_data_length = 251である場合、転送最大データ = 244（251-4-3）
- MAX_ATT_MTU = 156 かつ le_data_length = 251である場合、転送最大データ = 153（156- 3）

7.6.6 カスタム Bluetooth LE サービスを追加する方法

すべての Bluetooth LE アプリケーションにおいて、ソースコードまたはライブラリの中で提供されている既存のサービスと並行して、カスタムのサービスを追加可能です。CPU1 が受信したすべての GAP/GATT イベントは、すべての Bluetooth LE サービスを初期化して GATT イベントを登録された Bluetooth LE サービスに転送するサービス･コントローラ（\Middlewares\ST\STM32_WPAN\ble\svc\Src の svc_ctl.c）に送られます。フローの例を 図 24：心拍数プロジェクト - ミドルウェアとユーザアプリケーションの相互作用 に示します。

サービスごとに custom_xxx.c と custom_xxx.h が必要となります。

ユーザに提供されるパブリック･インタフェースは 3つのみ存在する必要があります。

void Custom_xxx_Init (void)

これは custom_xxx.c に実装され、次の処理を行います。

- サービスの生成とキャラクタリスティックの追加を行います。
- API SVCCTL_RegisterSvcHandler() を用いてサービス･コントローラにコールバックを登録します。

Custom_xxx_Init() をコールするために、アプリケーションには関数 SVCCTL_InitCustomSvc() が実装されている必要があります。

SVCCTL_RegisterSvcHandler() で登録されたコールバックは、サービス·コントローラから GATT イベントを受信するために使用されます。コールバックのタイプは、SVCCTL_EvtAckStatus_t (*SVC_CTL_p_EvtHandler_t)(void *p_evt) である必要があります。

Bluetooth LE サービスの定義によっては、受信した GATT イベントが custom_xxx.c モジュールのみで処理することも可能ですが、多くの場合は、通知 Custom_xxx_Notification() によってアプリケーションに転送される必要があります。それぞれの GATT イベントは、1つのみの Bluetooth LE サービスに関連しています。サービス·コントローラが登録されたすべての Bluetooth LE サービスをコールして受信イベントを報告しないようにするために、コールバックは、GATT が処理済みであるのか無視されているのかをサービス·コントローラに返します。

戻り値には次の3種類があります。

1. SVCCTL_EvtNotAck : GATT イベントは Bluetooth LE サービスに無関係であったことを示します。サービス·コントローラは、ACK を受信するまで、登録されたその他の Bluetooth LE サービスに対してこの GATT イベントの報告を繰り返します。登録されたすべての Bluetooth LE サービスから GATT イベントに対する確認応答を受けない場合には、通知 SVCCTL_App_Notification() を用いてアプリケーションに報告されます。
2. SVCCTL_EvtAckFlowEnable : GATT イベントは処理済みであり、サービス·コントローラは、登録されたその他の BLE サービスにもアプリケーションにもそのことを報告しないことを示します。
3. SVCCTL_EvtAckFlowDisable : GATT イベントには確認応答済みであり、サービス·コントローラは、登録されたその他の Bluetooth LE サービスにもアプリケーションにもそのことを報告しないことを示します。ただし、GATT イベントは処理されていません。サービス·コントローラは、このイベントを破棄してはならないことをトランスポートレイヤに通知します。その場合、コマンド hci_resume_flow() がコールされるまで、トランスポートレイヤはそれ以上イベントを報告しません。フローが再開されるとすぐに、確認応答を受けていないイベントは再度報告されます。それらは、それ以上報告されないすべての Bluetooth LE ユーザ hci イベントであり、GATT イベントを確認応答していない Bluetooth LE サービスに対するもののみではないことに注意してください。

```
tBleStatus Custom_xxx_UpdateChar( Custom_xxx_ChardId_t ChardId, uint8_t *
p_payload )
```

この API は、サーバのキャラクタリスティックを更新するためにアプリケーションによって使用されます。インタフェースの ChardId と Bluetooth LE スタックに送信される UUID とのマッピングは、Bluetooth LE サービスの中に実装されている必要があります。

```
void Custom_xxx_Notification( Custom_xxx_Notification_t *p_notification )
```

この API は、該当する場合に、Bluetooth LE サービスが受信する GATT イベントのアプリケーションへの報告に使用されます。

参考資料

8 HCI レイヤ･インタフェース上への Bluetooth LE アプリケーションの構築

CPU2 は Bluetooth LE HCI レイヤのコプロセッサとして使用することができます。その場合、独自の HCI アプリケーションを実装するか、あるいは既存のオープンソース Bluetooth LE ホストスタックを使用する必要があります。

多くの Bluetooth LE ホストスタックでは、UART インタフェースを使用して Bluetooth LE HCI コプロセッサとの通信を行います。STM32WB デバイスの等価物理レイヤは、セクション 13.2 : メールボックス･インタフェースに記載されているメールボックスです。

メールボックスは、Bluetooth LE とシステムチャネル両方へのインタフェースを備えています。Bluetooth LE ホストスタックは、メールボックスの Bluetooth LE チャネルに送信されるコマンドバッファを構築し、メールボックス経由で受信するイベントをレポートするインタフェースを備えている必要があります。メールボックス Bluetooth LE ホストスタックの適応に加えて、非同期パケットのリリース時にはメールボックス･ドライバに通知を行う必要があります。

システムチャネルは Bluetooth LE ホストスタックによる操作は行われません。独自のトランスポート･レイヤを実装し、メールボックス･ドライバに送信されるシステムコマンドバッファを構築して、メールボックスから受信するイベント（メールボックス･ドライバに対する非同期バッファのリリース通知を含む）を管理する必要があります。あるいは、システムコマンドバッファを構築する 提供済みトランスポート･レイヤの上にインタフェースを提供するメールボックス拡張ドライバ（セクション 13.3 : メールボックス･インタフェース - 拡張に記載）を使用して、システム非同期イベントを管理する必要があります。

セクション 13.2 : メールボックス･インタフェースに記載されているようにメールボックスを使用して Bluetooth LE HCI レイヤのコプロセッサの上にアプリケーションを構築する例として、BLE_TransparentMode プロジェクトを使用することもできます。

9 Thread

9.1 概要

CPU2 コアに搭載されているThread スタックは、Thread ネットワーク・プロトコルのオープンソース実装である OpenThread から提供され、Nest によってリリースされています。

OpenThread は、スタック内の異なるレベルでさまざまなサービスに対応するいくつかの API を提供しています。それらの API（STM32WB ファームウェア・パッケージに資料が添付されています）は、すべて CPU1 コアにエクスポートされており、アプリケーションから直接使用できます。

STM32WB ファームウェア・パッケージには、シンプルな Thread アプリケーションの動作方法をデモンストレーションする例が複数含まれています。これらのアプリケーションを実行するには、適切な CPU2 ファームウェア・バイナリをダウンロードする必要があります。

表 28 にある詳細説明のように、3種類の主要な MO ファームウェアが使用可能です。

表 28. Thread に使用可能な MO ファームウェア

CPU2 ファームウェア・ライブラリ	機能	コメント
stm32wb5x_Thread_FTD_fw.bin	FTD : フル Thread デバイス	デバイスは、境界ルータ（リーダ、ルータ、エンドデバイス、スリーピーエンドデバイス）を除くすべての Thread ロールをサポート可能です。 Thread ロールについては、セクション 13.10.5 : に説明されています。
stm32wb5x_Thread_MTD_fw.bin	MTD : 最小 Thread デバイス	デバイスは、「エンドデバイス」または「スリーピーエンドデバイス」としてのみ振る舞うことができます。 FTD 設定に対する MTD 設定の利点は、必要なメモリが少ないことです。
stm32wb5x_BLE_Thread_fw.bin	静的同時モード	デバイスは、1つのバイナリの中に 2つのスタック（Bluetooth LE と Thread）が搭載されています。

9.2 起動方法

Thread を起動する最も簡単な方法は、次の2つのアプリケーションを使用することです。

- Thread_Cli_Cmd :
  CLI コマンド経由で Thread スタックを制御する方法を示します。UART 経由で HyperTerminal（PC）からボードに CLI（コマンド・ライン・インタフェース）コマンドが送信され、シンプルなユースケースの生成に使用できます。これは、認証テスト（Thread GRL テスト・ハーネス）の実行に使用されるアプリケーションです。
- Thread_Coap_Generic : P-NUCLEO-WBxx ボードが2枚必要です。他のボードと CoAP メッセージを交換するボードを示します。このアプリケーションでは、片方のデバイスはリーダーとして、もう一方はエンドデバイスまたはルータとして振る舞います。

関連する readme.txt ファイルとともに、これら2つのアプリケーションが STM32WB ファームウェア・パッケージの中に含まれています。

9.3 Thread 設定

いずれかの Thread アプリケーションを開始する前に、次のことを実行する必要があります。

- 適切なファームウェア（Thread MTD、Thread FTD、Thread 静的モード）をダウンロードします。
- 正しいオプションバイトを使用します。

図 37. ユーザオプションバイトの設定

注意： **OpenThread スタックは、さまざまな設定を行うためのコンパイル･フラグをいくつか用意しています。ただし、STM32WB 内部のスタックはバイナリとして提供されているため、それらのフラグは固定されており、ユーザは変更できません。選択したフラグは、表 29 に示されているファイルで見ることができます。**

表 29. Thread 設定用ファイル

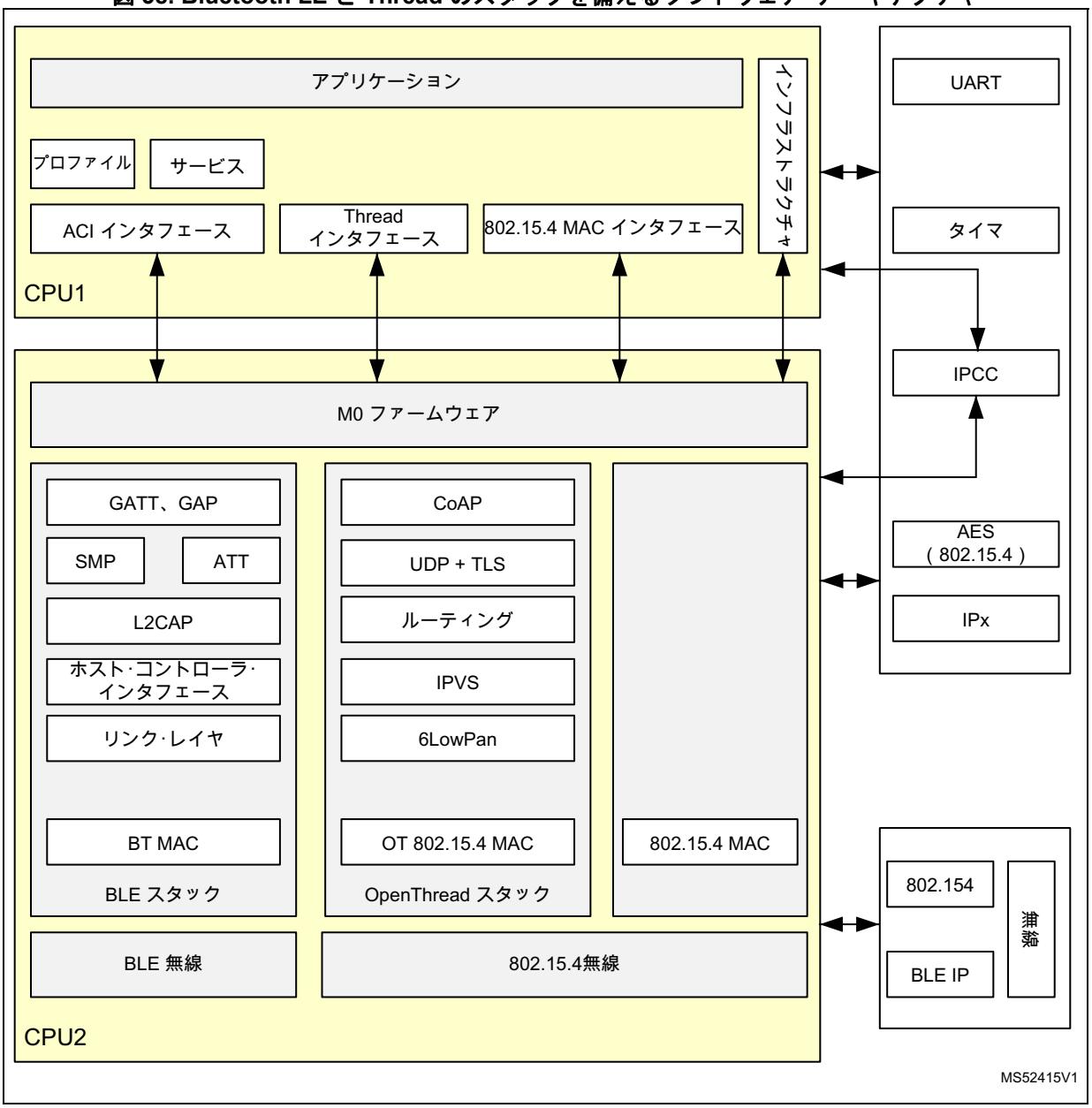
ファイル名	設定
openthread_api_config_ftd.h	Thread FTD CPU2 ファームウェアとともに使用します。
openthread_api_config_mtd.h	Thread MTD CPU2 ファームウェアとともに使用します。
openthread_api_config_concurrent.h	静的同時モード CPU2 ファームウェアとともに使用します。

Thread アプリケーションのビルド時には、ダウンロードした CPU2 ファームウェアに応じて適切な設定ファイルを使用する必要があります。この設定ファイルの中のフラグを使用して、どの API をエクスポートして CPU1 アプリケーションが使用できるようにするかを定義する必要があります。すでに示したように、これらのフラグは変更しないでください。

9.4 アーキテクチャの概要

図 38 は、Bluetooth LE と Thread の2つのスタックを有するソフトウェア全体のアーキテクチャを示します。CPU2 上で動作するすべてのコードは、バイナリ·ライブラリとして提供されます。ユーザがアクセスする必要があるのは CPU1 コアのみであり、CPU2 で動作するファームウェアはブラックボックスに見えます。ACI インタフェースと Thread インタフェースの両方とも、それぞれ Bluetooth LE と Thread タスクにアクセス可能です。

図 38. Bluetooth LE と Thread のスタックを備えるソフトウェア·アーキテクチャ



9.5 コア間通信

すべての OpenThread API は CPU1 に公開されており、CPU2 で動作するスタックの制御に使用可能です。STM32WB ミドルウェアは、2つのコア間の通信を管理します。

アプリケーションが OpenThread 関数をコールすると、同期メッセージが IPCC 経由で CPU2 に送信されます。この関数に関連付けられたパラメータは、共有メモリに格納されます。

OpenThread 関数コールは、システム全体が同期されたままであること保証するために、コマンドが完了するまでホールドされます（図 39 参照）。アプリケーションは、特定のイベントで通知を受けるコールバックの登録が可能です。図 40 に示されているように、これらの通知もホールドされます。

図 39. OpenThread 関数コール

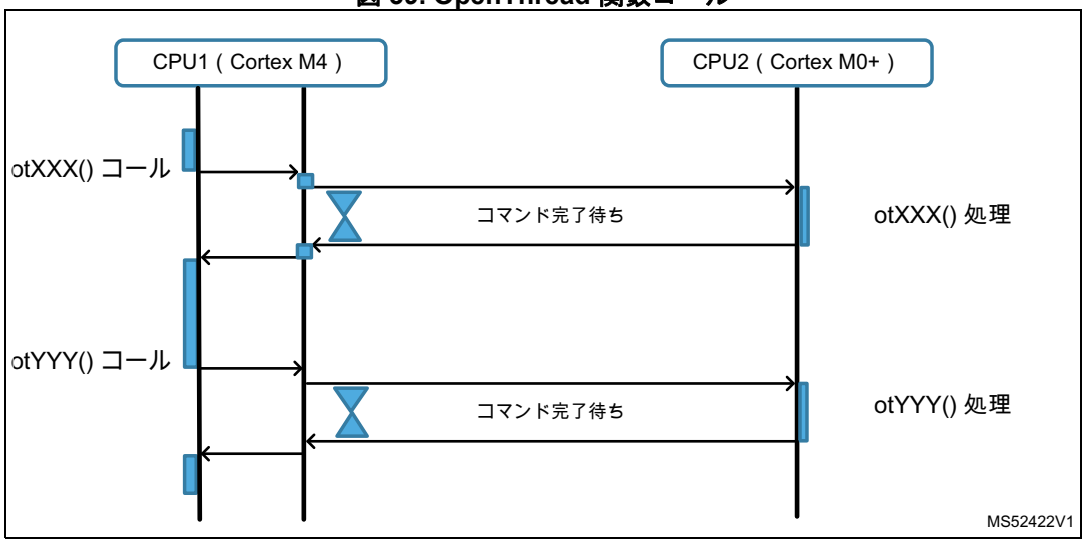
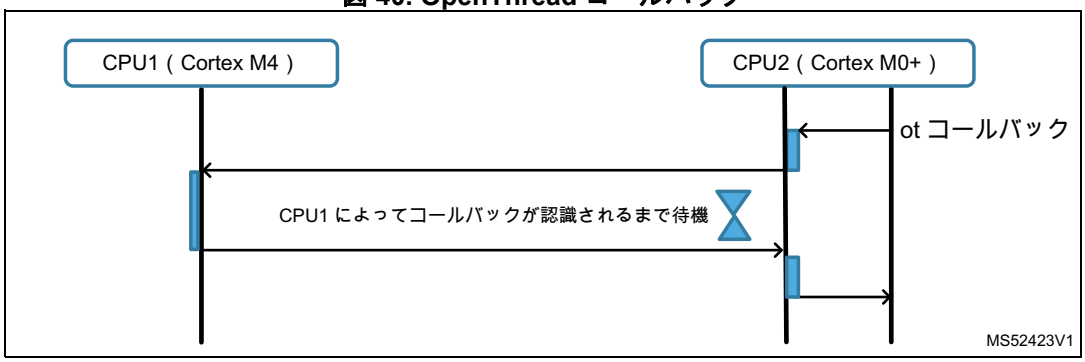


図 40. OpenThread コールバック



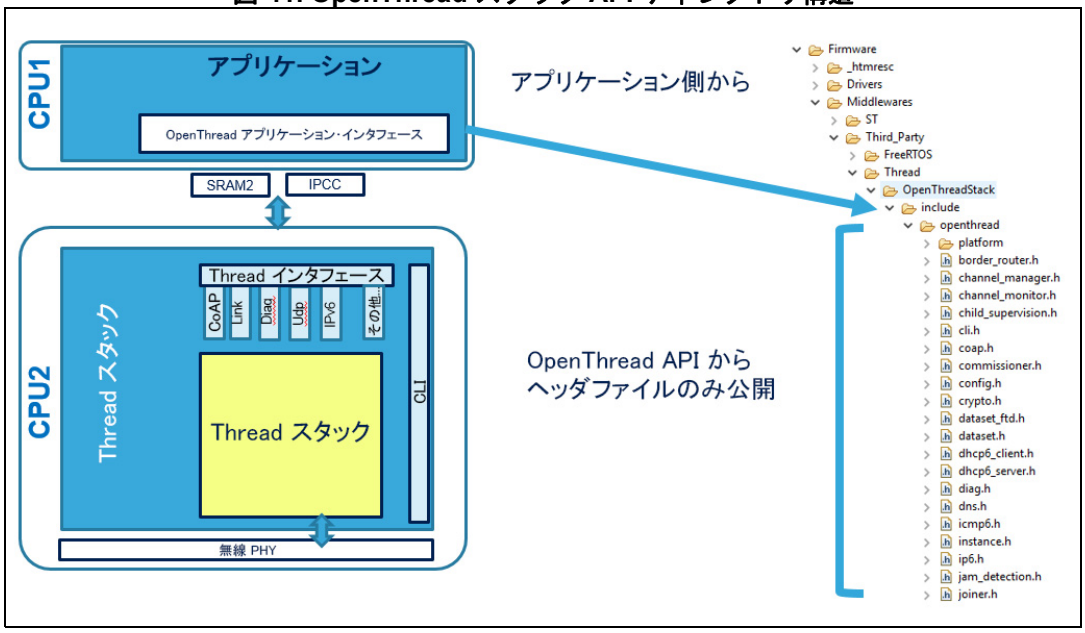
9.6 OpenThread API

OpenThread には、スタック内の異なるレベルでさまざまなサービスに対応するいくつかの API が提供されています。

- CoAP サービスの管理に使用される関数：otCoapStart()、otCoapSendRequest()
- UDP データグラムの管理に使用される関数：otUdpOpen()、otUdpConnect()
- 無線設定を管理する関数：otLinkSetChannel()
- IPV6 アドレスを管理する関数：otIp6AddUnicastAddress

使用可能な関数は合計で 300以上あります。これらの API は、STM32WB ファームウェア･パッケージに含まれている STM32WBxx_OpenThread_API_User_Manual.chm に記載されています。

図 41. OpenThread スタック API ディレクトリ構造



9.7 OpenThread API の使用方法

OpenThread API は、システムが単一プロセッサ上で動作しているかのように使用できます。Thread インタフェースによってすべてのマルチコア･メカニズム（IPCC、共有メモリ）が隠蔽されますので、CPU2 で動作する OpenThread スタックに CPU1 がアクセス可能となります。

ただし、次のサブセクションに記載されているように、STM32WB が OpenThread インタフェースを実装する方法に関連して、2つの特殊な点が存在します。

9.7.1 OpenThread インスタンス

OpenThread API の多くは、以下の関数 otThreadSetEnabled() の例ではボールド体で示されている、OpenThread インスタンスを定義するパラメータ aInstance を入力として使用します。

otThreadSetEnabled(otInstance ***aInstance**, bool aEnabled)

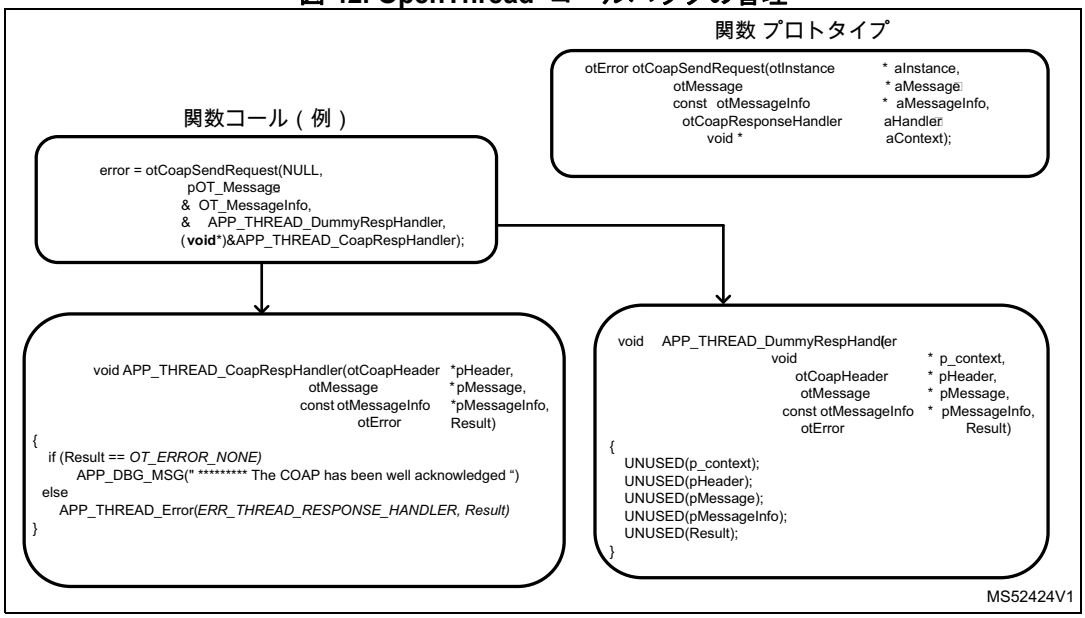
STWM32WB Thread 実装において、OpenThread インスタンスは CPU2 ファームウェアの先頭に直接割り当てられています。CPU1 はこのパラメータを扱う必要はなく、常に NULL にセットされています（以下のコード･フラグメントのボールド体参照）。

```
error = otThreadSetEnabled(NULL, true);
  if (error != OT_ERROR_NONE)
  {
    APP_THREAD_Error (ERR_THREAD_START,error);
  }
}
```

9.7.2 OpenThread コールバックの管理

STWM32WB Thread 実装において、OpenThread　関数の中でパラメータとして渡されるコールバックは、標準OpenThread　関数のプロトタイプに厳密には従っていません。これは、デュアルコア･アーキテクチャの制約によるものです。図 42 に示すように、アプリケーション･コールバックはコンテキスト･パラメータの中で渡される必要があります。

図 42. OpenThread･コールバックの管理



注：　OpenThread･コールバックが管理されている方法を確認する最も簡単な方法は、STM32WB　ファームウェア提供物の中の各種アプリケーションを参照することです。

9.8 Thread･アプリケーションのためのシステム･コマンド

コマンドの中には、Thread･アプリケーションからコール可能なものもあります。

- SHCI_C2_THREAD_Init() : Thread ･スタックを起動します。初期化フェーズの最後にコールされます。
- SHCI_C2_FLASH_StoreData() : 不揮発性Thread･データを Flash メモリに格納します。データを Flash メモリに格納する必要があるタイミングを決定するのはアプリケーションとなります（コミッション･フェーズの後、ネットワーク設定の後など）。

注 : この操作には数秒かかる場合がありますので、Thread が動作していない場合に限りコールする必要があります。

- SHCI_C2_FLASH_EraseData() : 不揮発性Thread･データを Flash メモリから消去します。

注 : この操作には数秒かかる場合がありますので、Thread が動作していない場合に限りコールする必要があります。

- SHCI_C2_CONCURRENT_SetMode() : 同時モードでの CPU2 上の Thread 動作を有効化または無効化します。
- SHCI_C2_RADIO_AllowLowPower() : 802_15_4 無線 IP が低消費電力モードに移行するのを許可または禁止します。
- SHCI_GetWirelessFwInfo() : ロードされているワイヤレスバイナリに関係する情報を読み出します。

9.8.1 不揮発性 Thread･データ

Thread 仕様に従って、いくつかの値を後で再使用するために Flashメモリに格納する必要があります。これらの値は、次のエンティティに関係しています。

1. アクティブ動作データセット :

   新しいアクティブ動作データセットを受信した場合に必ず書き込まれます。これは、コミッショナまたはその他の外部エンティティがアクティブ動作データセットを更新した場合のみ発生します。

2. ペンディング動作データセット :

   新しいペンディング動作データセットを受信した場合に必ず書き込まれます。これは、コミッショナまたはその他の外部エンティティがペンディング動作データセットを更新した場合のみ発生します。

3. ネットワーク情報 :

   デバイスの役割が変化した場合に必ず書き込まれます（取外し、子、ルータ、リーダなど）。MAC や MLE フレームカウンタが特定の閾値を超えてインクリメントされた場合に必ず書き込まれます。

4. 親情報 :

   子が親にアタッチされた場合に必ず書き込まれます。

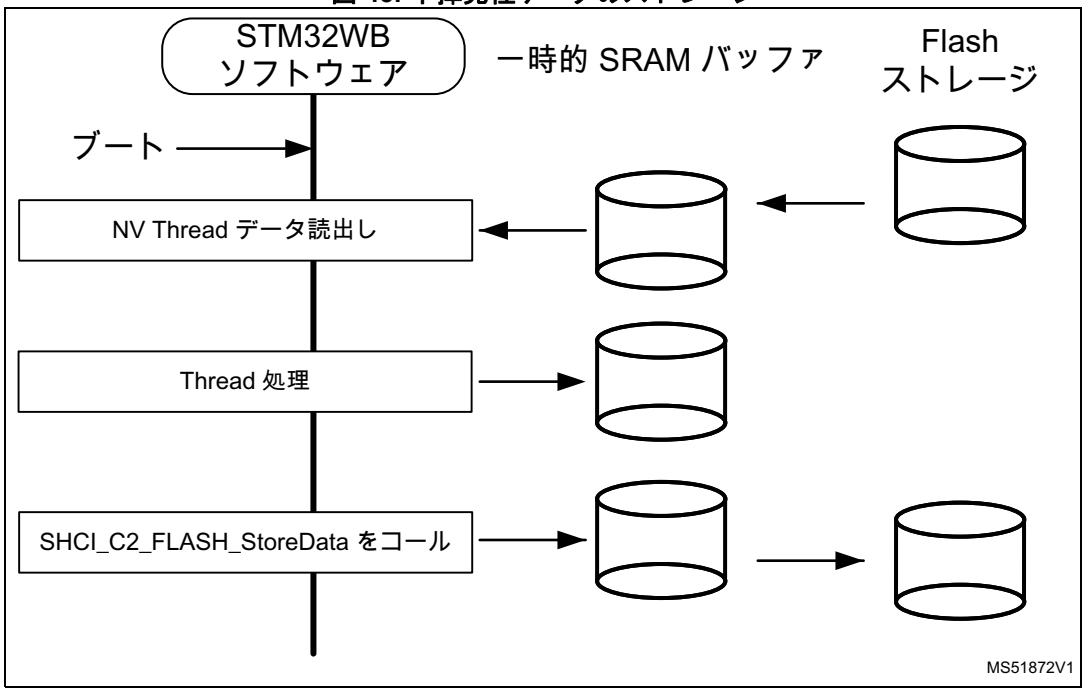
5. 子情報 :

   子が子テーブルに追加されたり削除されたりした場合に必ず書き込まれます。

リセット後、不揮発性 Thread･データセットは Flash メモリから自動的に読み出されます。実行時は、OpenThread がこの不揮発性データを内部 SRAM バッファに定期的に格納と更新を行います（図 43 参照）。関数 SHCI_C2_FLASH_StoreData() を使用してこの不揮発性データの Flash メモリへのコピーを強制するかどうかはアプリケーション次第です。この操作によって Flash メモリへのアクセスがブロックされる（CPU に対しても同様）ため、リアルタイム制約が存在しない場合（Thread 停止後など）に行う必要があります。

注： **関数 SHCI_C2_FLASH_StoreData() は、otInstanceReset () または otInstanceFactoryReset に対するコールの後に自動的にトリガされます。**

図 43. 不揮発性データのストレージ



9.8.2 低消費電力サポート

消費電力を最小とするには、デバイスを SED（スリーピーエンドデバイス）モードとして、その親からのメッセージをポーリングしたりデータを送信したりするためにウェイクアップする必要があります。ほとんどの時間デバイスはスリープし、自動的に低消費電力モードに移行します。低消費電力 Thread デバイスは、バッテリ電源で何年間もスリープと動作を行うことができます。

システムが低消費電力モードにある場合、アプリケーションから otCmd が送信されると直ちにシステムはウェイクアップしてコマンドを実行した後に、低消費電力モードに戻ります。あるアプリケーションが複数の otCmd を連続送信した場合、システムは高頻度でウェイクアップしてスリープに戻ります。短時間で複数の不必要なウェイクアップ／スリープサイクルを行うリスクを防止するには、アプリケーションは、関数 SHCI_C2_RADIO_AllowLowPower() を通じて無線が低消費電力モードに移行するのを許可または禁止します。この関数の例は、Thread_SED_Coap_Multicast という名前のアプリケーションの中に適用されています。

10 OpenThread･アプリケーションのステップバイステップ設計

このセクションでは、STM32WB デバイスにOpenThread･アプリケーションを設計して実装する方法に関する情報とコード例について説明します。

10.1 初期化フェーズ

STM32WB Thread･アプリケーションの初期化には、いくつかの手順が必ず必要となります:

- デバイス（HAL、リセット･デバイス、クロックと電源設定）を初期化します。
- プラットフォーム（ボタン、LED など）を設定します。
- ハードウェア（UART、デバッグなど）を設定します。
- CPU2 を起動し、（CPU1 側の）アプリケーションにシステム通知を送信します。
- 通知を受信したら、アプリケーションは Thread 設定を開始します。

10.2 Thread･ネットワークのセットアップ

ファームウェア･パッケージで提供されている Thread･アプリケーションの中で、Thread･ネットワークのセットアップは、常に同じ関数 APP_THREAD_DeviceConfig() を用いて行われます。

この関数は次の手順を処理します。

- 永続的 Thread パラメータを消去してクリーンスタート otInstanceErasePersistentInfo() を実行します。
- ノードの役割が変更された場合（ノードがルータになった場合など）に OpenThread ･スタックからコールされるアプリケーション･コールバックを登録します。otSetStateChangedCallback() を使用して操作を行います。
- チャネルを設定します：otLinkSetChannel()
- PANID をセットします：otLinkSetPanId()
- IPv6 通信を有効にします：otIp6SetEnabled()
- CoAP サーバを開始します：otCoapStart()
- CoAP リソースを CoAP サーバに追加します：otCoapAddResource()
- Thread･プロトコル操作を開始します：otThreadSetEnabled()

これらの手順の後

- この Thread･ネットワークで最初のボードである場合、そのノードはリーダ（Thread で緑 LED がオンになるなど）となります。
- ネットワーク上にリーダがすでに存在する場合、そのノードはルータまたは子として参加します（Thread で赤 LED がオンになるなど）。

参考資料

10.3 CoAP リクエスト

コンストレインド･アプリケーション･プロトコル（CoAP）とは、制約のあるノードおよびネットワーク（低消費電力、損失大など）とともに使用する特殊なウェブ転送プロトコルのことです。

CoAP は、アプリケーション･エンドポイント間のリクエスト／レスポンス相互作用モデルを提供し、サービスとリソースの内蔵ディスカバリをサポートし、URI やインターネット･メディア･タイプなどのウェブの主要概念を含んでいます。

注： **このセクションでは、CoAP に関連するすべてのOpenThread API の詳細説明は行いませんが、主要関数と、STM32WB Thread 例の一部として使用可能な抽象化の概要を提供します。**

10.3.1 otCoapResource の生成

```
typedef struct otCoapResource
{
    const char *           mUriPath; ///< URI パス文字列
    otCoapRequestHandler   mHandler; ///< 受信リクエストを処理するためのコールバック
    void *                 mContext; ///< アプリケーション固有コンテキスト
    struct otCoapResource *mNext;    ///< リスト上の次の CoAP リソース
} otCoapResource;
```

宣言は、次のとおりです。

```
static otCoapResource OT_Ressource = {C_RESSOURCE,
APP_THREAD_DummyReqHandler, (void*)APP_THREAD_CoapRequestHandler, NULL};
```

ここで：

`mUriPath = C_RESSOURCE` ： リソース名（「light」など）

`mHandler = APP_THREAD_DummyReqHandler()` ： ダミーのリクエスト･ハンドラ。（下記注記参照）

`mContext = APP_THREAD_CoapRequestHandler()` ： サーバが COAP リクエストを受信したときにコールされるハンドラ

`mNext = NULL` ： 未使用

注： **STM32WB の実装（OpenThread･インタフェース用ラッパーを備えたデュアルコア CPU1/CPU2）により、パラメータ mHandler はダミーのリクエスト･ハンドラ関数（なにも行わない）でフィルし、有効なリクエスト･ハンドラを mContext パラメータに渡す必要があります。**

10.3.2 CoAP リクエストの送信

アプリケーション Thread_Coap_Generic には、CoAP リクエストの送信を容易とする抽象化レイヤが提案されています。これは次の関数を使用して行われます。

```
static void APP_THREAD_CoapSendRequest(otCoapResource* pCoapRessource,
    otCoapType CoapType,
    otCoapCode CoapCode,
    const char *Address,
    uint8_t* Payload,
    uint16_t Size)
```

10.3.3 CoAP リクエストの受信

サーバが CoAP リクエストを受信したときにコールされます。

CoAP リクエスト･ハンドラのプロトタイプは次のとおりです。

```
static void APP_THREAD_CoapRequestHandler(otCoapHeader * pHeader,
    otMessage            * pMessage,
    const otMessageInfo  * pMessageInfo)
```

この関数において、次の関数をコールすることで受信メッセージの読出しが可能です。

```
otMessageRead()
```

（otCoapHeaderGetType() を使用して）メッセージのタイプが確認可能である場合、CoAP レスポンスを送信する必要があります（セクション 11.3：Thread_Coap_Genericなど参照）。

10.4 コミッショニング

コミッショニングには、コミッショナ･ロールのデバイスと、ジョイナ･ロールの別デバイスが必要です。コミッショナは、既存ネットワークの Thread･デバイスであるか、役割を行う Thread･ネットワーク外のデバイス（携帯電話など）であるかのいずれかです。

ジョイナとは、Thread･ネットワークへの参加を望んでいるデバイスのことです。

Thread･コミッショナは、ネットワーク上のデバイスを認証するために使用されます。マスタキーなどの Thread･ ネットワーク資格情報の転送は行いませんし、所有も行いません。

この資料は、外部コミッショナや境界ルータなしの基本的なオンメッシュ･コミッショニングを対象としています。

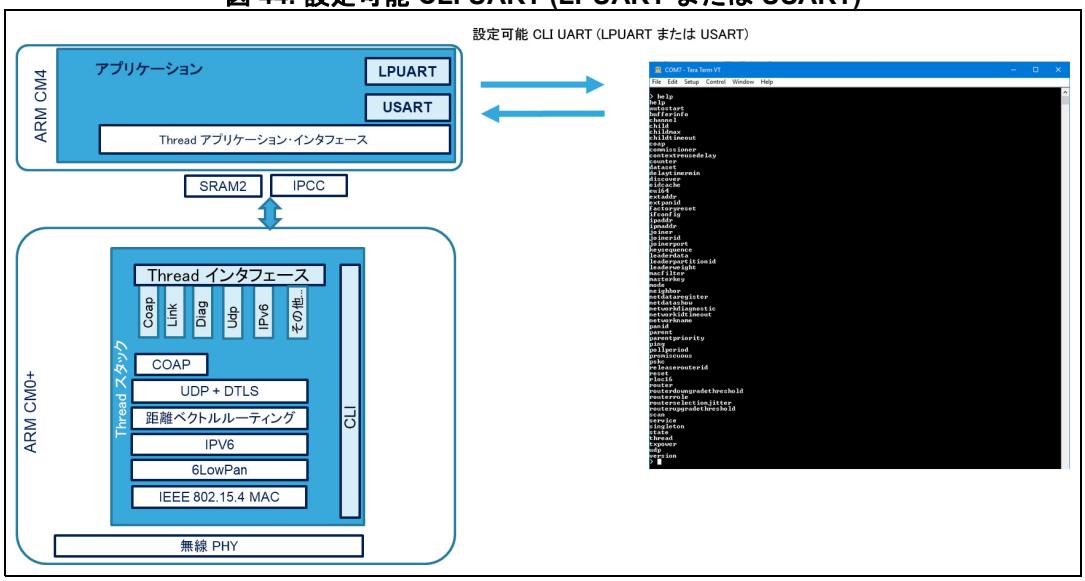
Thread_Commissioning アプリケーションは、シンプルなコミッショニングの例をデモンストレーションするものです。

10.5 CLI

OpenThread･スタックは、コマンド･ライン･インタフェースを通じて設定と管理の API を公開しています。

認証環境（GRL テスト･ハーネス）は、CLI を使用してテストケースを実行します。

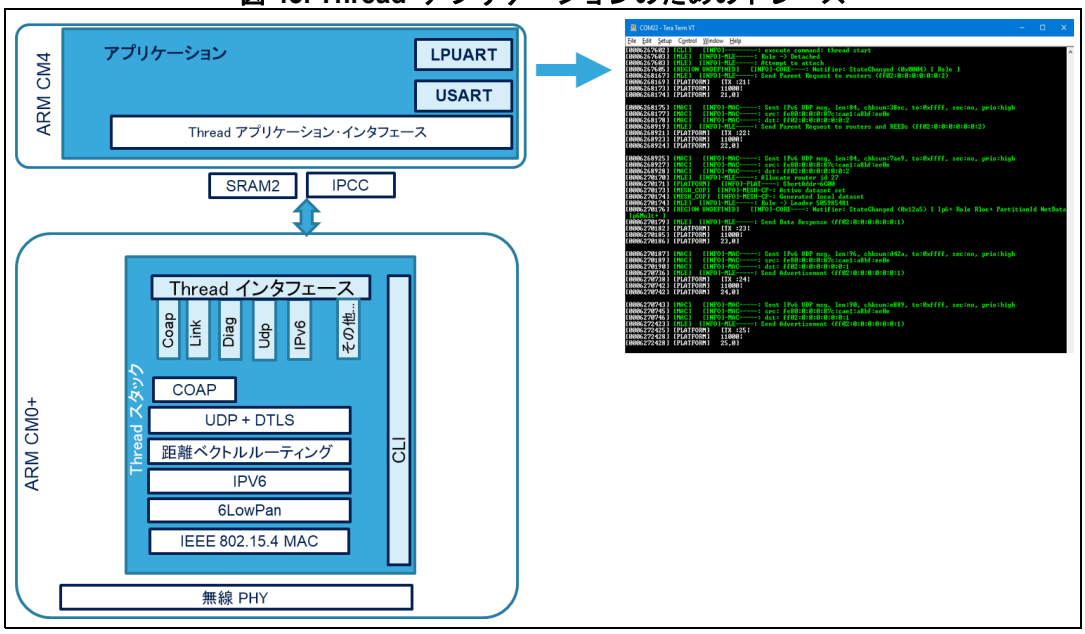
図 44. 設定可能 CLI UART (LPUART または USART)



10.6 トレース

CPU1アプリケーションと CPU2アプリケーションからのトレースは、UART にルーティングされます（app_conf.h ファイルを使用してコンパイル時に設定）。

図 45. Thread·アプリケーションのためのトレース



OpenThread·スタックのトレース·レベルは、otSetDynamicLogLevel() を使用してOpenThread API 経由で動的に設定可能です。

参考資料

11 STM32WB OpenThread･アプリケーション

11.1 Thread_Cli_Cmd

このアプリケーションは、コマンドライン経由で OpenThread･スタックを制御する方法を示します。

CLI コマンドは、UART 経由で HyperTerminal（PC）から STM32WBxx_NUCLEO ボードに送信されます。

これは、認証プロセスに使用されるアプリケーションです。

11.2 Thread_Coap_DataTransfer

このアプリケーションは、CoAP メッセージング･プロトコルを使用してデータの大きなブロックを転送する方法をデモンストレーションするものです。

このアプリケーションでは、メッシュローカル･スコープとマルチキャスト･アドレッシング･タイプを使用して、ファイル転送先である子デバイスのメッシュローカル IP アドレスを検査します。

ノードは、ルータとエンドデバイスの 2つの転送ロールに分類されます。

デバイスを 2台使用するこのアプリケーションでは、片方はリーダ（ルータ）として、もう一方はエンドデバイス（子モード）として振る舞います。

2枚のボードのリセット後、そのうち 1枚はリーダ･モード（緑の LED2が ON）、もう 1枚は子モード（赤の LED3が ON）となります。

1台のデバイスで子モードが確立されると、マルチキャスト･モードでプロビジョニング手順を開始し、リーダ･デバイスの IP アドレスを検査します。

さらに、これを使用してユニキャスト･モードでファイル転送手順を開始し、操作が成功すると青 LED が点灯して合図します。

11.3 Thread_Coap_Generic

このアプリケーションは、CoAP メッセージの使用をデモンストレーションするものです。CoAP マルチキャスト･リクエストの送信に抽象レベルを提供します。

このアプリケーションには STM32WB ボードが2枚必要です。目的は、CoAP メッセージを互いに交換する両方のボードをデモンストレーションすることです。このアプリケーションでは、片方のボードはリーダとして、もう一方はエンドデバイスまたはルータとして振る舞います。

11.4 Thread_Coap_Multiboard

このアプリケーションは、CoAP を使用して複数のボードにユニキャストでメッセージを送信する方法を示します。STM32WBxx_NUCLEO ボードを2枚から 5枚使用するように設計されています。

このアプリケーションの目的は、小さな Thread･メッシュ･ネットワークを作成することです。

セットアップが正しく設定されていれば、CoAP リクエストが次の順番で1枚のボードから次のボードに自動的かつ連続的に転送されます。

- ボード 1 → ボード 2 → （...）→ ボード n → ボード 1 → …

各ボードには特定の IPv6 アドレスが関連付けられています。

- ボード 1には fdde:ad00:beef:0:442f:ade1:3fc:1f3a
- ボード 2には fdde:ad00:beef:0:442f:ade1:3fc:1f3b
- ボード 3（存在する場合）には fdde:ad00:beef:0:442f:ade1:3fc:1f3c
- ボード 4（存在する場合）には fdde:ad00:beef:0:442f:ade1:3fc:1f3d
- ボード 5（存在する場合）には fdde:ad00:beef:0:442f:ade1:3fc:1f3e

11.5 Thread_Commissioning

このアプリケーションは、コミッショナとジョイナの間のコミッショニング･プロセスをデモンストレーションするものです。

コミッショニング･プロセスを使用して自分の Thread･パラメータ（チャネル、PANID、マスタキー）を別のデバイスに配布するデバイスが示されています。

このアプリケーションには STM32WBxx_NUCLEO ボードが2枚必要です。

片方のデバイスはコミッショナとして、もう一方はジョイナとして振る舞います。

このアプリケーションでは、コミッショナは新規デバイスを自分の Thread･ネットワークに受け入れます。

11.6 Thread_FTD_Coap_Multicast

このアプリケーションは、フル･Thread･デバイスに対する CoAP マルチキャスト･メッセージの使用をデモンストレーションするものです。

注： **Thread FTD CPU2 バイナリとともに使用します。**

このアプリケーションはデバイスを 2台使用し、片方はリーダ（ルータ）として、もう一方はエンドデバイス（子モード）として振る舞います。

2枚のボード（それぞれ A、B とします）のリセット後、1枚はリーダ･モード（緑の LED2 が ON）、もう 1枚は子モード（赤の LED3 が ON）となります。

ボード A からボード B に CoAP コマンドを送信するには、ボード A の SW1 押ボタンを押します。ボード B が CoAP コマンドを受信してその青の LED1 を点灯します。同じボタンをもう一度押すと、青の LED1が消灯します。

同じ CoAP コマンドをボード B からボード A に送信することができます。

11.7 Thread_SED_Coap_Multicast

このアプリケーションは、スリーピーエンドデバイスから送信される CoAP マルチキャスト･メッセージの使用をデモンストレーションするものです。

注： **Thread MTD CPU2 バイナリとともに使用します。**

提案のユースケースにはボードが2枚必要となります。

- 片方のボードは FTD モードでリーダ（ルータ）として振る舞います（ボード A）。
- もう一方のボードは MTD モードで SED として振る舞います（ボード B）。

リーダとして振る舞うボードでは､FTD アプリケーションを Flash 書込みする必要があります｡CPU2 にアプリケーション「Thread_FTD_Coap_Multicast」とThread FTD バイナリを使用してください。

SED として振る舞うボードでは、CPU2 に MTD アプリケーション「Thread_SED_Coap_Multicast」と Thread MTD バイナリを Flash 書込みする必要があります。

2枚のボードのリセット後、片方のボード（A）は自動的にリーダ･モード（緑の LED2が ON）となり、もう一方（B）は数秒後に SED モード（赤の LED3が ON）となります。

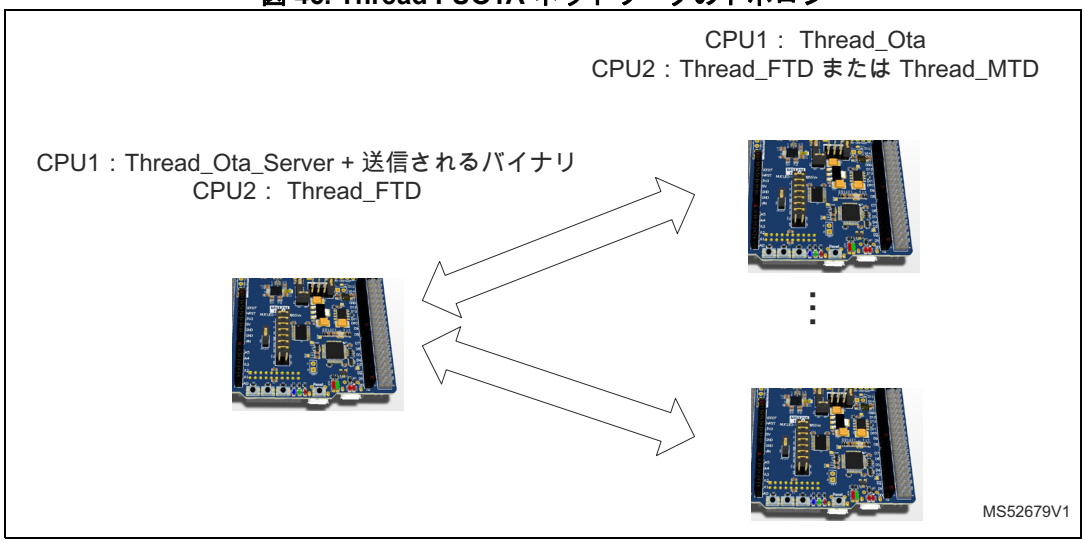
この段階では、これら 2枚のボードは同一の Thread ネットワークに属しており、デバイス 2がデバイス 1に CoAP マルチキャスト･リクエストを毎秒送信し、その青色 LED を点滅させます。

11.8 Thread FUOTA

11.8.1 原則

目的は、Thread プロトコルを使用してリモート･デバイス上の CPU1 アプリケーション･バイナリまたは CPU2 ワイヤレス･コプロセッサ･バイナリを更新することです。

図 46. Thread FUOTA ネットワークのトポロジ



この Thread には、特定のアプリケーションで Thread プロトコルが動作している STM32WBxx ボードが最低2枚必要となります（図 46参照）。

- Thread_Ota_Server アプリケーションが動作するボード 1枚
- Thread_Ota アプリケーションが動作するボード 1枚以上

FUOTA プロセスは、1デバイスのみで同時に実行可能です。

サーバは FUOTA プロビジョニング･プロセスを開始し、1つのクライアントがそれに応答する必要があります。複数のクライアントが、その時点で 1つ更新されます。

11.8.2 メモリマッピング

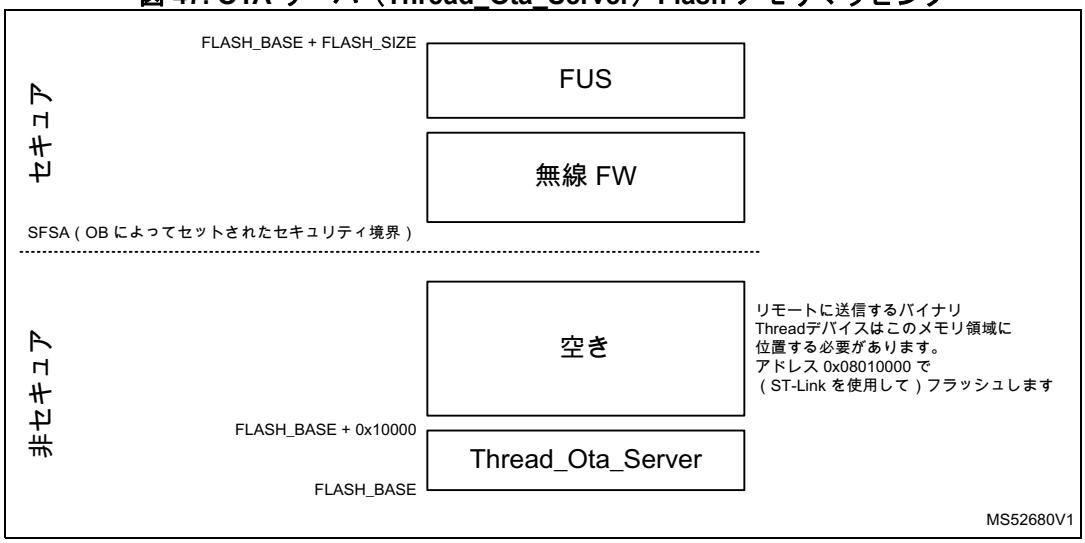
サーバ側

（CPU1 か CPU2 かどちらかの更新のために）リモート･デバイスにインストールされるバイナリ･ファイルは、サーバ側の「空き」領域メモリに最初に Flash 書込みされる必要があります（図 47参照）。

転送されるバイナリの最大サイズは次の値に等しくなります。

空き領域サイズ = SFSA アドレス - （FLASH_BASE - 0x8010000）

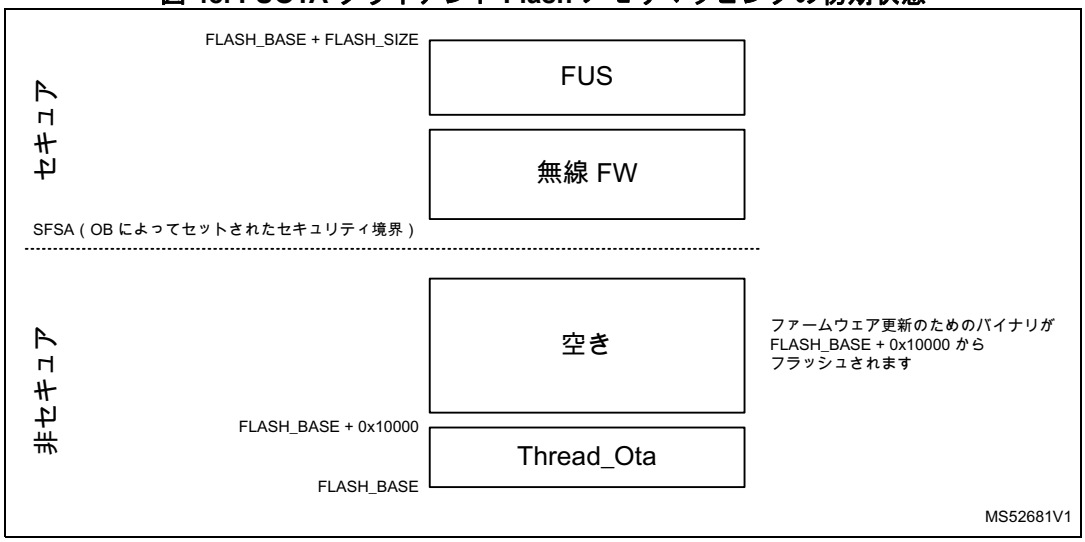
図 47. OTA サーバ（Thread_Ota_Server）Flash メモリマッピング



クライアント側

クライアント側では、バイナリをサーバから受信する前には Flash メモリは図 48 に示すようになっています。

図 48. FUOTA クライアント Flash メモリマッピングの初期状態



バイナリデータをサーバ側から受信した後に、CPU1 バイナリ転送と CPU2 バイナリ転送に対して、Flash メモリはそれぞれ図 49 と図 50 に示したように更新されます。

図 49. CPU1 バイナリ転送後の FUOTA サーバ Flash メモリマッピング

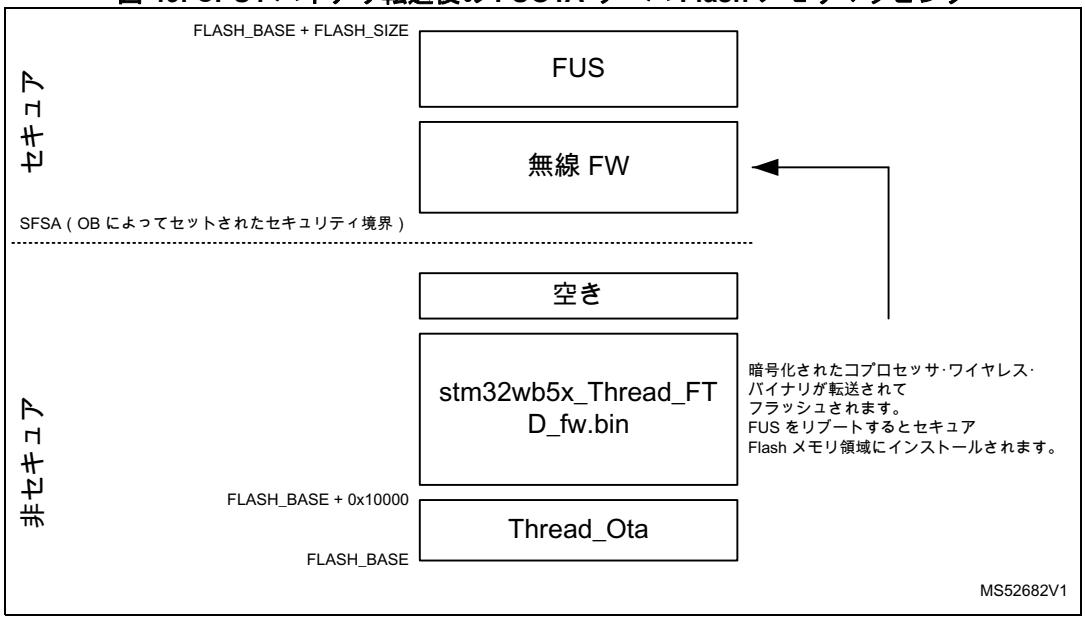
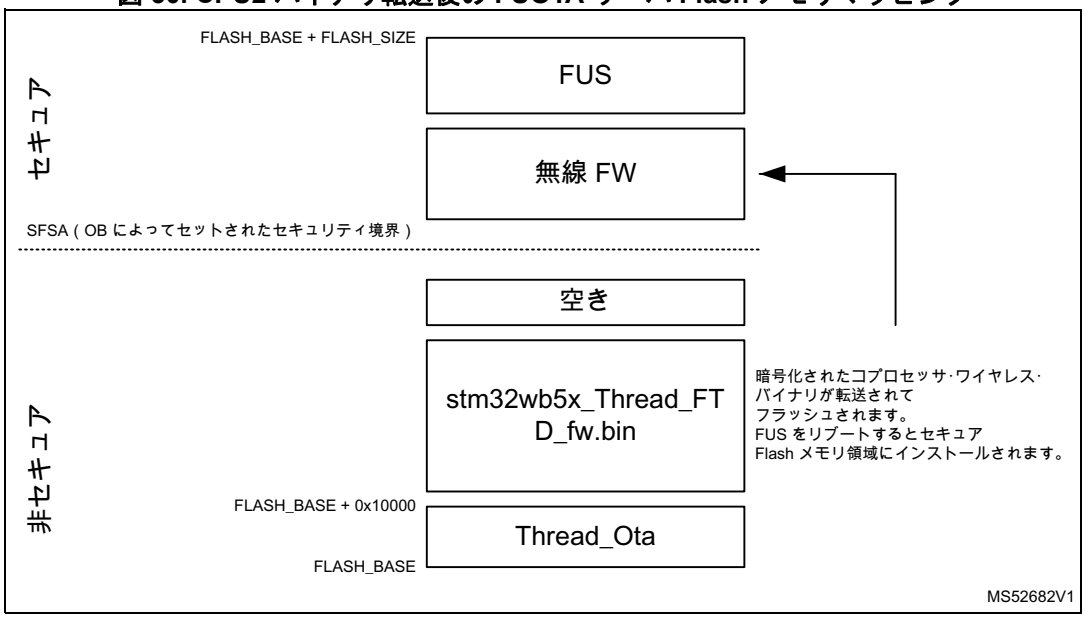


図 50. CPU2 バイナリ転送後の FUOTA サーバ Flash メモリマッピング

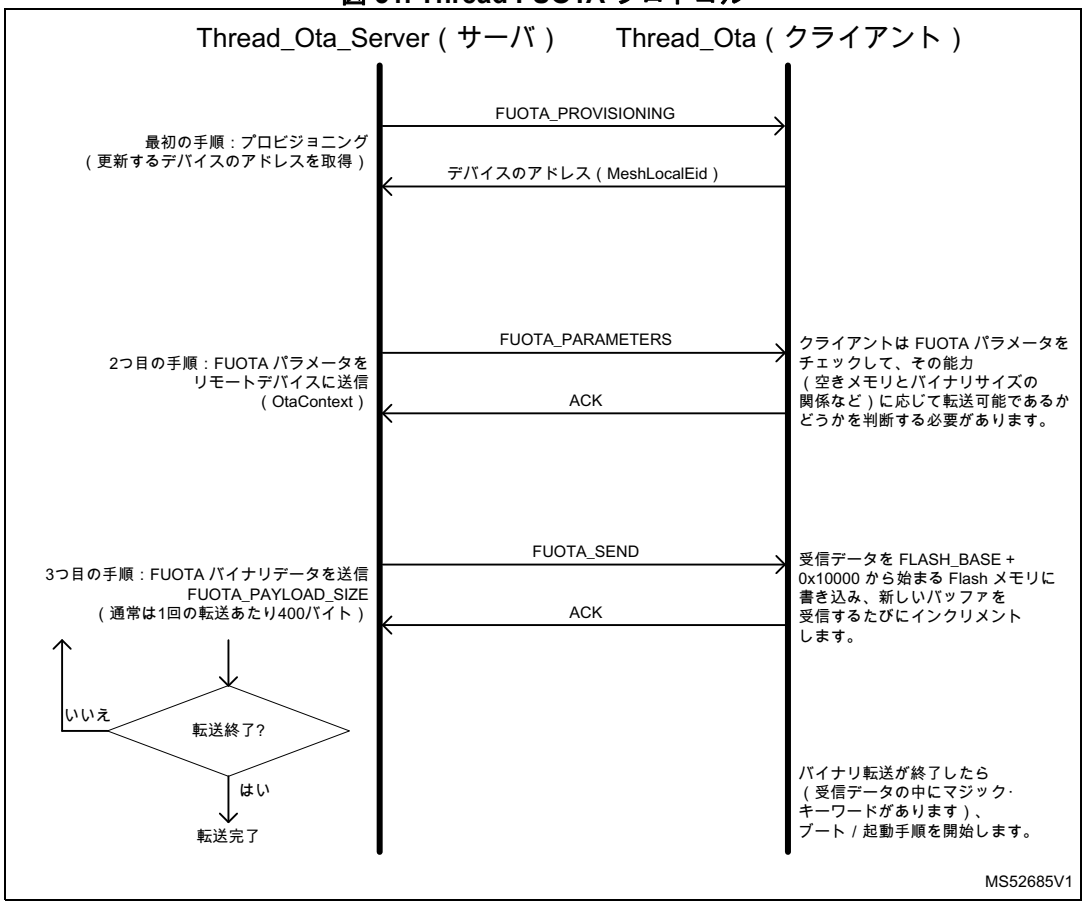


11.8.3 Thread FUOTA プロトコル

これは、CoAP リクエストに基づく Thread を使用して CPU2 ワイヤレス･コプロセッサ･バイナリまたは CPU1 FW アプリケーションを更新するための STMicroelectronics 独自プロトコルです。

ファームウェア更新転送の手順の詳細を図 51 に示します。

図 51. Thread FUOTA プロトコル



1. サーバは、メッセージを送信して、FUOTA が処理されるリモートデバイスのアドレスを記録します。
   サーバは、リソースに対するマルチキャスト、確認不可、Get CoAP リクエストである「FUOTA_PROVISIONING」を送信します。
   リモートデバイスは、ネットワーク･トポロジから独立した Thread･インタフェースを識別する Mesh Local Eid（Endpoint IDentifier）を応答します。
2. OtaContext データ構造がリモートデバイスに送信されます。その中には次の情報が含まれています。
   - ファイルタイプ：FW_APP 更新または FW_COPRO_WIRELESS 更新
   - バイナリサイズ：
     転送するバイナリのサイズ（バイト単位）
   - ベースアドレス：バイナリデータのコピー先リモートデバイスの Flash メモリの開始アドレス
   - マジック･キーワード：バイナリの最後を指定するキーワード

3. リモートデバイスにバイナリを転送します。
転送は、FUOTA_PAYLOAD_SIZE（デフォルト 400バイト）のバッファを用いて行われます。この値は、サーバ側で設定可能です。
転送のたびに、Thread_Ota_Server は次のバッファ転送を続ける前にリモートデバイスからの確認応答を待ちます。

リモート側では、受信したデータバッファごとに Flash メモリに書き込まれます。

マジック･キーワードが見つからない場合、それが送信する最後のバッファであることを意味します。

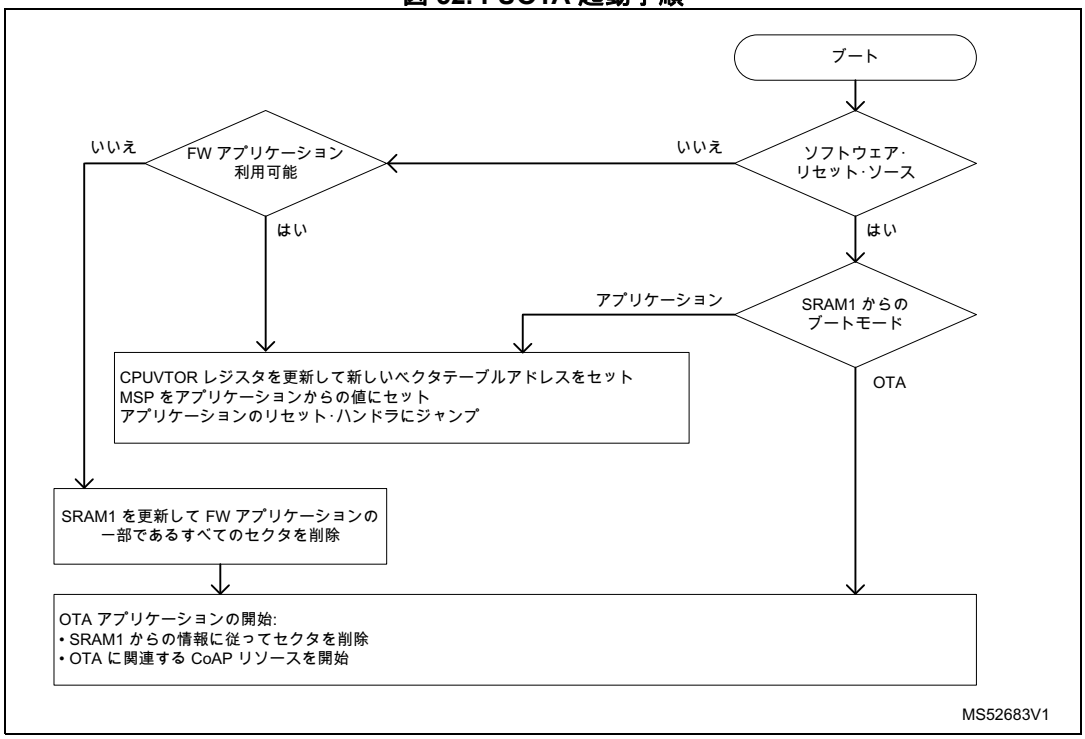
11.8.4 FUOTA アプリケーション起動手順

バイナリデータがリモートデバイス（Thread FUOTA クライアント）に転送された後の起動手順は CPU1 アプリケーションの更新と CPU2 コプロセッサ･ワイヤレス･バイナリの更新とで異なります。

CPU1 に対する FUOTA

クライアント側でバイナリ転送が完了すると、図 52 に示す処理が行われて OTA 専用アプリケーション（Thread_Coap_Generic_Ota など）にジャンプします。

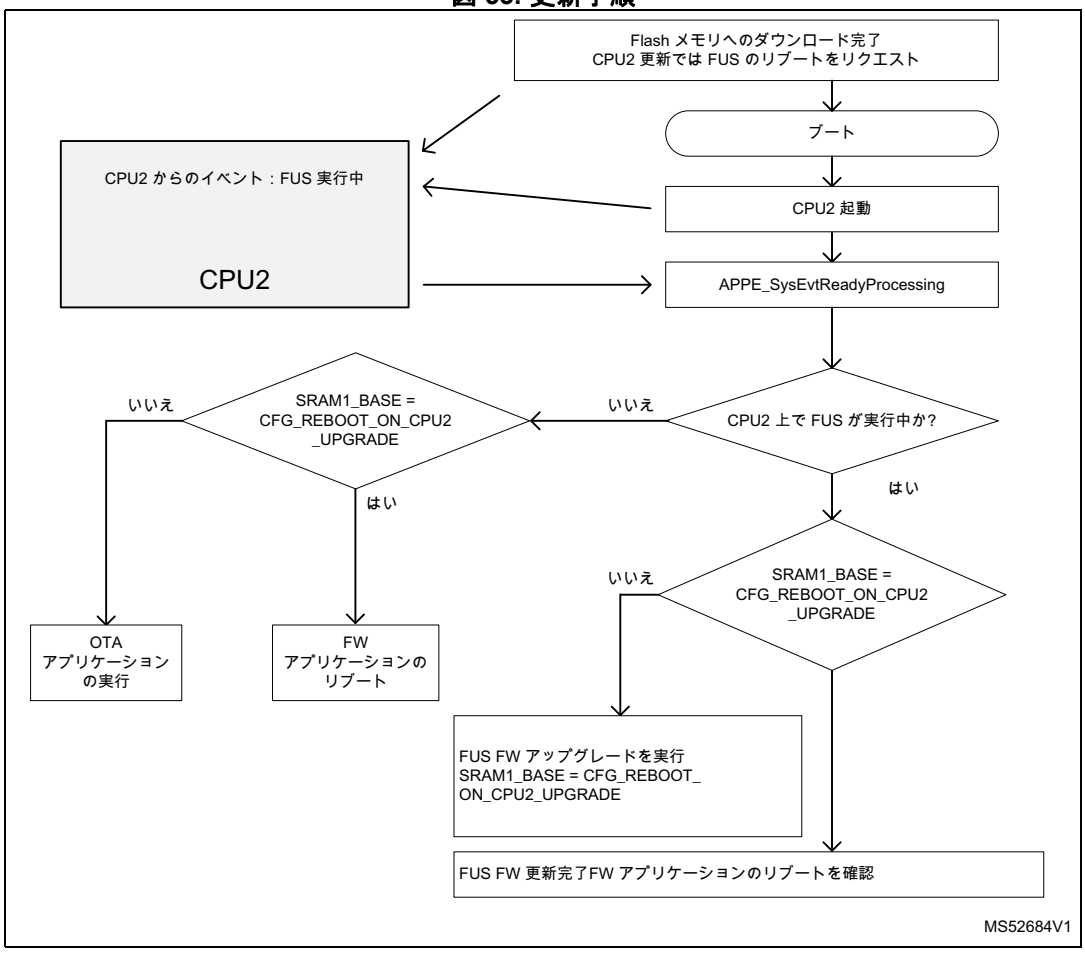
図 52. FUOTA 起動手順



CPU2 に対する FUOTA

CPU2 の更新には、セキュア･バイナリの復号化とインストールを処理する FUS（ファームウェア更新サービス）ソフトウェア･コンポーネントが必要となります。

図 53. 更新手順



11.8.5 アプリケーション

Thread_Ota_Server

このアプリケーションは、FUOTA サーバとして振る舞う STM32WB 1Nucleo ボードにロードされる必要があります。

Thread_Ota

このアプリケーションは､FUOTA クライアントとして振る舞う STM32WB Nucleo ボードにロードされる必要があります。

Thread_Coap_Generic_Ota

このアプリケーションは、Thread_Coap_Generic とほぼ同一であり、以下が相違点となります。

- 次の特殊タグを使用します（エンドデータ転送とデータの一貫性の管理のため）。
  - TAG_OTA_END：マジック･キーワード値は thread_ota アプリケーションの中でチェックされます。
  - TAG_OTA_START：マジック･キーワード･アドレスは、バイナリイメージの先頭から 0x140 の位置にマッピングされる必要があります。

  したがって、0x140 の位置のメモリ内容を読み出した値はマジック･キーワード値と等しい必要があります。
- 上記のセクションを配置するには、スキャッタファイルを更新する必要があります。

  IAR の例：

```
Vector table and ROM start @ moved to 0x08010000:
define symbol __ICFEDIT_intvec_start__ = 0x08010000;
define symbol __ICFEDIT_region_ROM_start__ = 0x08010000;
define region OTA_TAG_region   = mem:[from
(__ICFEDIT_region_ROM_start__ + 0x140)   to
(__ICFEDIT_region_ROM_start__ + 0x140 + 4)];
keep { section TAG_OTA_START};
keep { section TAG_OTA_END };
place in OTA_TAG_region   { section TAG_OTA_START };
place in ROM_region   { readonly, last section TAG_OTA_END };
```

12 MAC IEEE Std 802.15.4-2011

12.1 概要

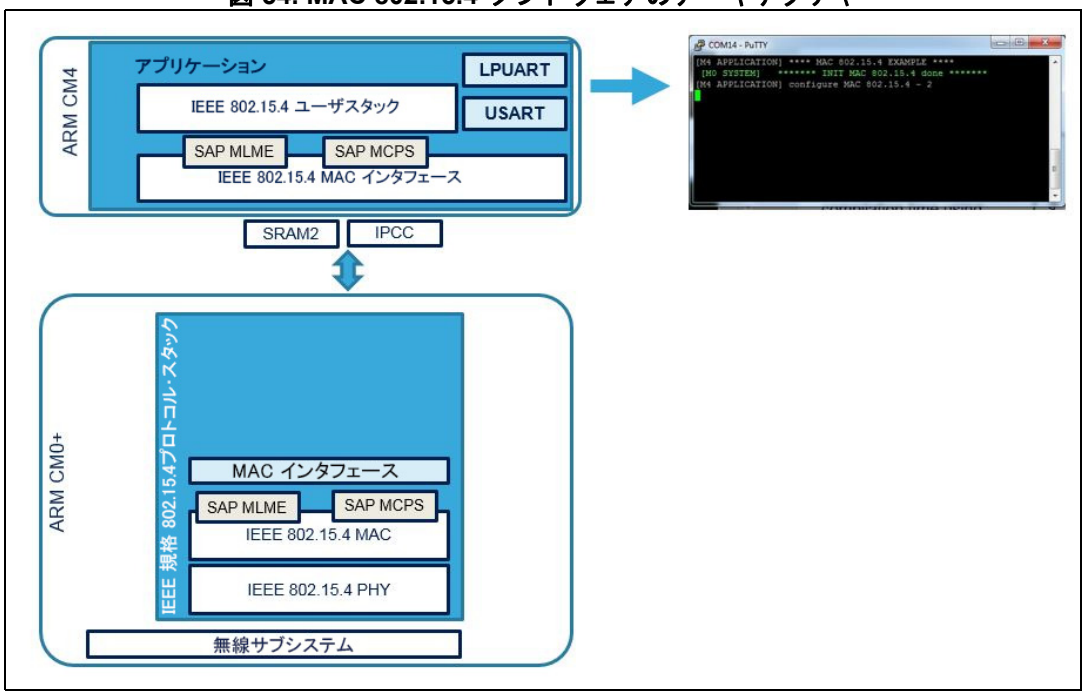
MAC IEEE Std 802.15.4-2011レイヤは、CPU2 コア（無線プロトコル・プロセッサ）上で動作する MAC 専用ファームウェアによって実装されています。MAC レイヤは、RF サブシステム・コンポーネントを処理する PHY レイヤに依存します。

この実装はバイナリ形式で提供されていて CPU2 上で動作するものであるため、MAC サービス・アクセス・ポイントをアドレス指定可能であるように、MAC API は CPU1 コアに公開されています。そのため、自分の STM32 デバイスを FFD（フル機能デバイス：コーディネータ）としてセットアップすることも、IEEE Std 802.15.4-2011仕様書に記載されているような RFD（限定機能デバイス：ノード）としてセットアップすることも可能です。

12.2 アーキテクチャ

アプリケーション・プロセッサにカスタムのソリューションまたはサードパーティ製ソリューションを統合して、組織内 802.15.4ネットワークを実装する場合に使用される MAC ソフトウェアのアーキテクチャを図 54に示します。

図 54. MAC 802.15.4 ソフトウェアのアーキテクチャ

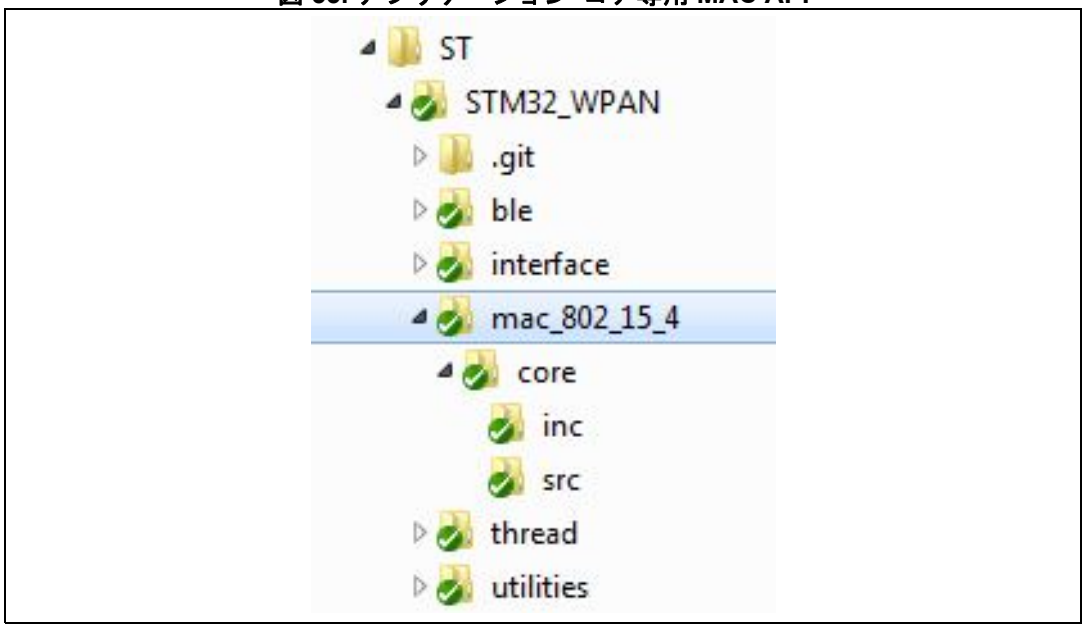


12.3 API

MAC IEEE Std 802.15.4-2011仕様書には、802.15.4ネットワーク･レイヤとメディア･アクセス･コントロール･レイヤの間のインタフェースが定義されています。この API を使用すると、MLME（MAC サブレイヤ管理エンティティ）と呼ばれる MAC 管理エンティティ･サービスを、MCPS（MAC 共通部分サブレイヤ･エンティティ）と呼ばれる MAC データ･サービスとして扱うことが可能となります。

アプリケーション･コア専用の MAC API とそれに対応した実装は、\Middlewares\ST\STM32_WPAN\mac_802_15_4 にあるミドルウェアから利用可能です（図 55 参照）。

図 55. アプリケーション･コア専用 MAC API



この実装は、STM32WB FW パッケージの Firmware\Middlewares\ST\STM32_WPAN\mac_802_15_4 ディレクトリにある STM32WBxx_MAC_802_15_4_User_Manual.chm に記載されています。詳細なプリミティブの説明は、IEEE Std 802.15.4-2011 文書からアクセス可能です。

12.4 起動方法

12.4.1 ボード設定

オプションバイトが図 56 にあるとおりにセットされていることを確認してください。

図 56. MAC 802.15.4のオプションバイト設定

12.4.2 MAC 無線プロトコル･プロセッサ CPU2 ファームウェア

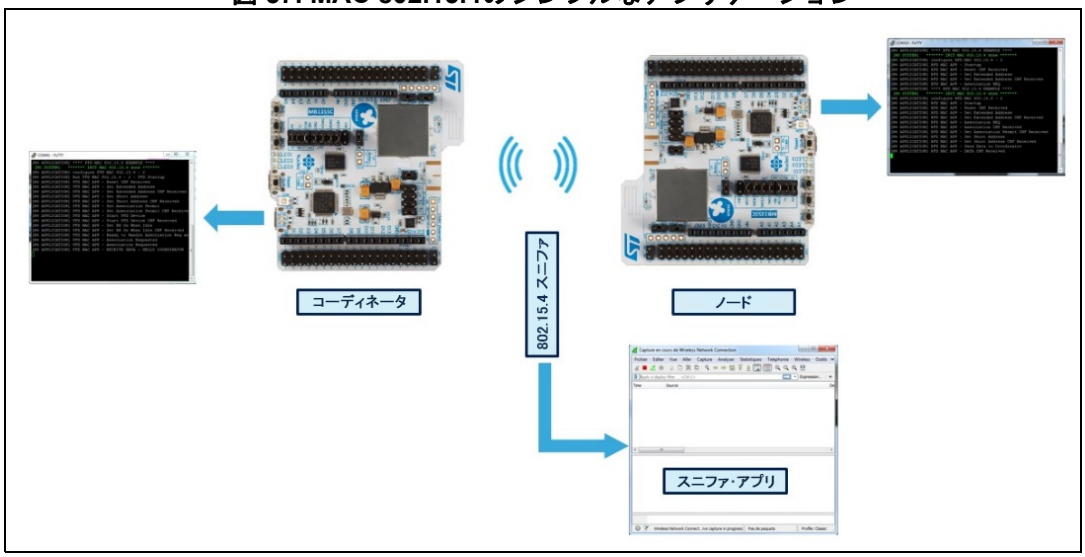
まず、CPU2 無線プロトコル･コアに適した専用 MAC ファームウェア･バイナリをダウンロードする必要があります。STM32WB FW パッケージの Firmware\Projects\STM32WB_Copro_Wireless_Binaries ディレクトリにある Release_Notes.html を参照してください。

12.4.3 MAC アプリケーション･プロセッサ･ファームウェア

カスタムのスタック･ソリューションの実装前や、CPU1 アプリケーション･コア MAC API 用に提供されたサードパーティ製スタックの統合前に、
STM32WB ボード2枚で同時に実行する必要のある以下の2つのアプリケーションを参照しながら MAC アプリケーション例を用いて、学ぶことができます。

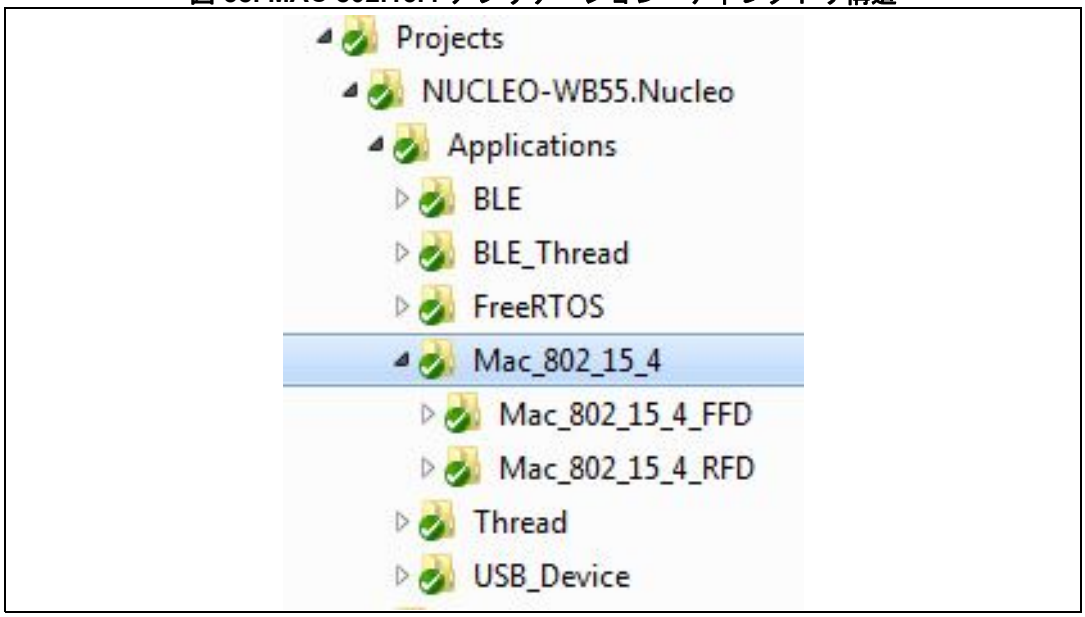
- Mac_802_15_4_FFD : シンプルな 802.15.4コーディネータの実装方法を示します。このデバイスは、アソシエーション･リクエストとしてネットワークを管理し、ノード要求に応じてデータの取得または提供を行います。
- Mac_802_15_4_RFD : シンプルな 802.15.4ノードの実装方法を示します。このデバイスは、コーディネータにアソシエーション･リクエストを送信します。アドレス指定されたコーディネータがリクエストに肯定的に応答すると、ノードはその新しいショートアドレスを受信し、データをコーディネータに送信します。

図 57. MAC 802.15.4のシンプルなアプリケーション



どちらのアプリケーションも Nucleo STM32WB ボード専用であり、NUCLEO-WBxx.Nucleo application Mac_802_15_4 ディレクトリにあります（図 58 参照）。

図 58. MAC 802.15.4 アプリケーション - ディレクトリ構造



readme.txt ファイルに、それぞれの 802.15.4デバイスで処理される MAC シーケンスが記載されています。このファイルはそれぞれの root プロジェクトにあります。

12.4.4 出力

コーディネータによって管理されるネットワークにノードが登録されたら、正しいチャネルに OTA スニファを使用して、アソシエーション·フェーズ中の 2枚のボードの間のネゴシエーションを聞くことも、データ交換を表示させることもできます。

アプリケーション·トレースは UART にルーティングされます。実装されているそれぞれの仮想 COM ポートに好みのターミナル·エミュレータを使用してハイパーターミナルのセッションを開始し、MAC ステップごとに確認を行えます。

コンソールを接続するための TTY セッション設定を以下に示します。

- ボーレート：115200
- データビット：8
- ストップビット：1
- パリティ：なし
- フロー制御：XON/XOFF

2つのアプリケーションを動作させると、図59～61 に示したハイパーターミナル画面のようになります。

図 59. コーディネータの起動

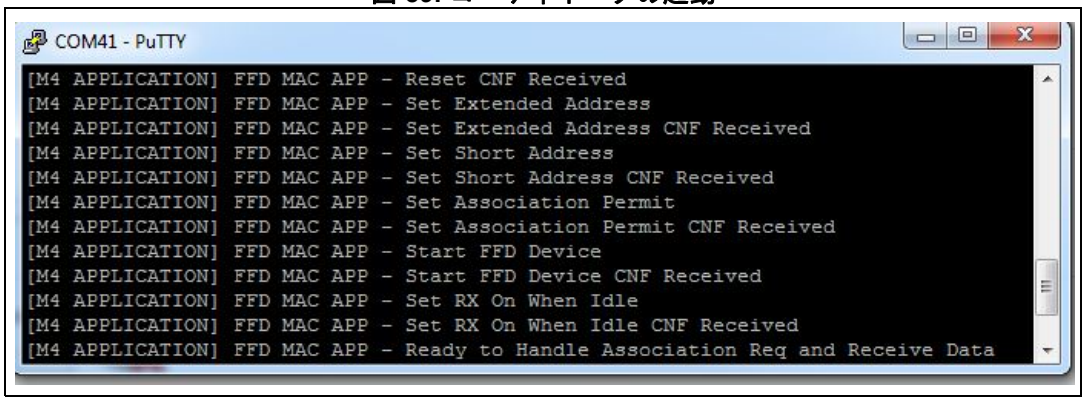


図 60. ノードの起動、アソシエーションのリクエスト、データ送信

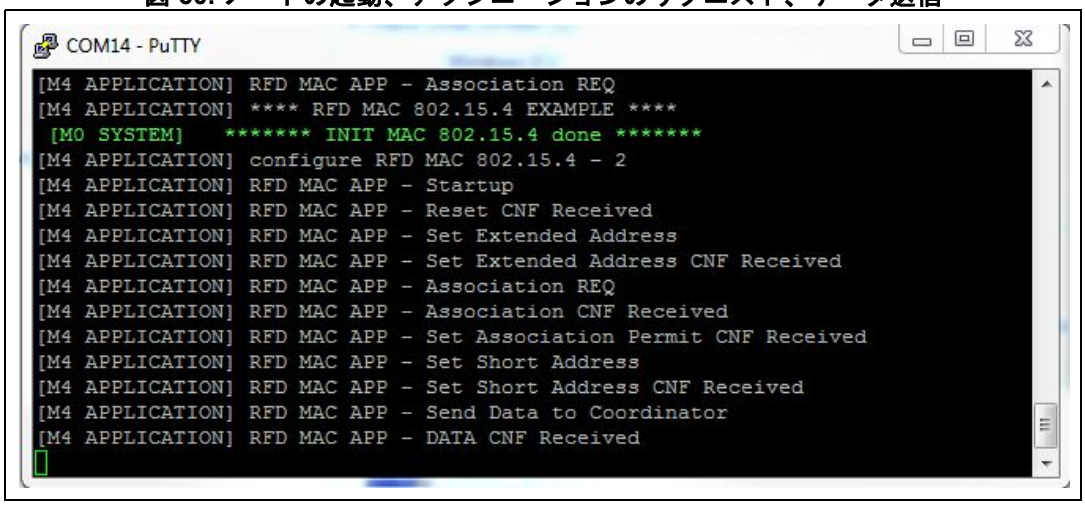


図 61. アソシエーション･リクエストとデータを受信するコーディネータ

```
[M4 APPLICATION] FFD MAC APP - Association Requested
[M4 APPLICATION] FFD MAC APP - RECEIVE DATA : HELLO COORDINATOR
```

12.4.5 MAC IEEE Std 802.15.4-2011システム

これは、現在実装されている MAC システム･コマンドです。

SHCI_C2_MAC_802_15_4_Init() によって、無線プロセッサ（CPU2）上の MAC レイヤと RF サブシステムが起動します。

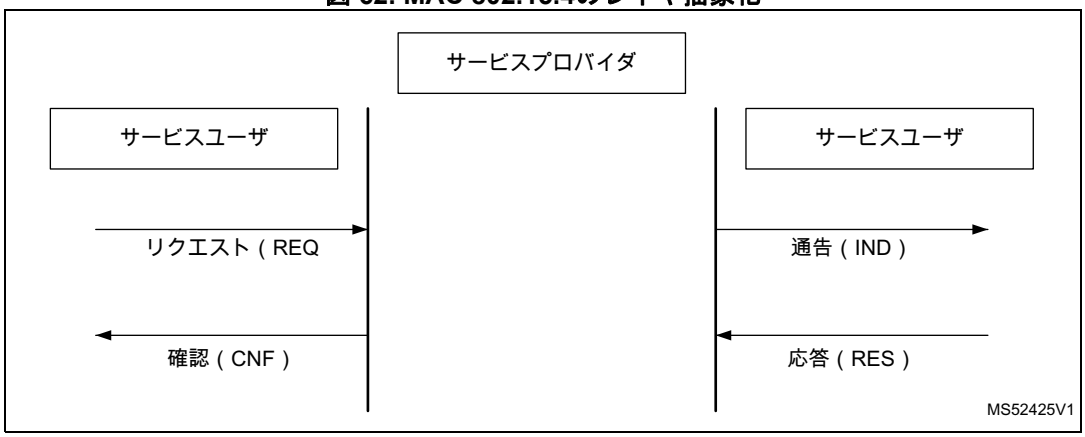
不揮発性データは、MAC レイヤによって保証されません。これらのデータが Flash メモリの中に保持されることを保証して、後で使用するために復元するのは、アプリケーション上位レイヤ次第です。

低消費電力機能はサポートされていません。

12.4.6 統合に関する推奨事項

MAC レイヤは、抽象化レイヤを実装することでサービス･プリミティブを提供します。MAC IEEE Std 802.15.4-2011仕様書に記載されているこの抽象化レイヤを図 62 に示します。

図 62. MAC 802.15.4のレイヤ抽象化



提案されている API を使用すれば、上位レイヤによって初期化された関連する定義済み構造体で REQ プリミティブと RES プリミティブをコールできます。MAC レイヤから通知を受けるには、MAC 通告（IND）または MAC 確認（CNF）と呼ばれるカスタムのコール関数を実装する必要があります。

リクエストとレスポンスの例

- 現在のデバイスのショートアドレスをセットします。
- セットするショートアドレスが格納されている初期化済み SetReq 構造体で MAC_MLMESetReq をコールします。

```
// Set Device Short Address
uint16_t shortAddr   = 0x1122;
SetReq.PIB_attribute = g_MAC_SHORT_ADDRESS_c;
SetReq.PIB_attribute_valuePtr = (uint8_t*) &shortAddr;
MacStatus = MAC_MLMESetReq ( &SetReq );
```

- アソシエーション通告に応答します。

アソシエーションがリクエストされた場合、コーディネータはショートアドレスでリクエスト側に応答を返すことができます。

- 属性付きショートアドレスを格納する初期化されたAssociateRes構造体でMAC_MLMEAssociateResを呼び出します。

```
APP_DBG("Srv task : Response to Association Indication");
  MAC_associateRes_t AssociateRes;
uint16_t shortAssociationAddr = 0x3344;
           memcpy
(AssociateRes.a_device_address,g_MAC_associateInd.a_device_address,0x08);
memcpy (AssociateRes.a_assoc_short_address,&shortAssociationAddr,0x08);
AssociateRes.security_level = 0x00;
AssociateRes.status = MAC_SUCCESS;

MacStatus = MAC_MLMEAssociateRes(&AssociateRes);
```

- 確認と通告の例

低位 MAC レイヤからの確認メッセージまたは通告メッセージの通知を受けるには、MAC_callbacks_t macCbConfig (example provided in app_ffd_mac_802_15_4.c) にカスタムのコールバックを登録する必要があります。

```
/* Mac Call Back Initialization */
macCbConfig.mlmeResetCnfCb = APP_MAC_mlmeResetCnfCb;
macCbConfig.mlmeScanCnfCb = APP_MAC_mlmeScanCnfCb;
macCbConfig.mlmeAssociateCnfCb = APP_MAC_mlmeAssociateCnfCb;
macCbConfig.mlmeAssociateIndCb = APP_MAC_mlmeAssociateIndCb;
....
```

- データ通告時のアクション

MAC サービスからデータを取得するには、カスタムのコールバックを実装する必要があります。

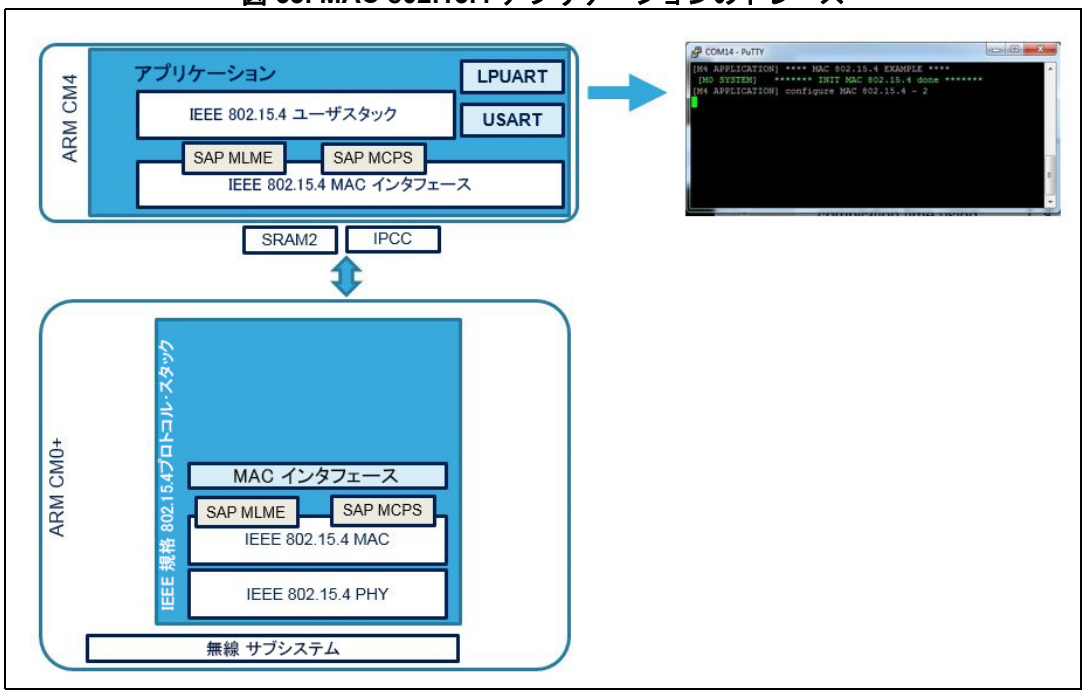
MAC レイヤからデータ通告メッセージを受信したら、macCbConfig.mcpsDataIndCb を使用して APP_MAC_mcpsDataIndCb コールバックをコールします。このコールバックは、MAC_dataInd_t 構造体（app_mac_802-15-4_process.c）に保持されている通告データを取得するために、次のようにして実装できます。

```
MAC_Status_t APP_MAC_mcpsDataIndCb( const  MAC_dataInd_t * pDataInd )
{

  memcpy(&g_DataInd,pDataInd,sizeof(MAC_dataInd_t));
  return MAC_SUCCESS;

}
```

図 63. MAC 802.15.4 アプリケーションのトレース



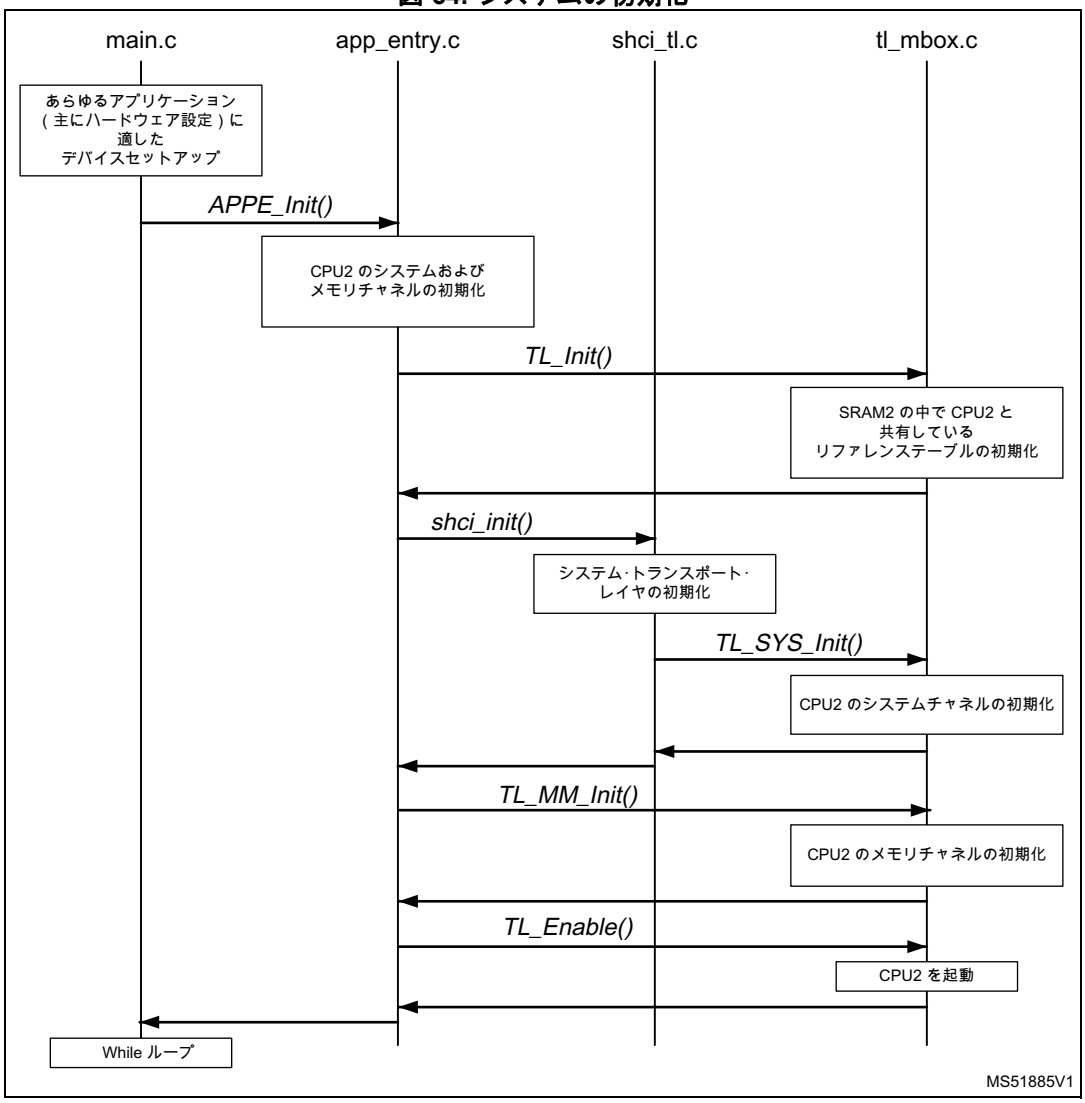
参考資料

13 付録

13.1 デバイス初期化詳細フロー

起動時に、まずデバイスが初期化された後に、CPU2 へのシステム·チャネルが初期化されます。このシーケンスが終了すると、CPU1 はバックグラウンド while ループに戻り、システムコマンドを受信する準備が整った CPU2 からの通知を待ちます。CPU1 は、RF（CPU2）には関係しない他のアプリケーションの初期化を実行することができます。この起動は、CPU2 が Bluetooth LE ホスト·スタック全体、HCI のみのインタフェース、OpenThread スタックのどれを実行していても同じです。

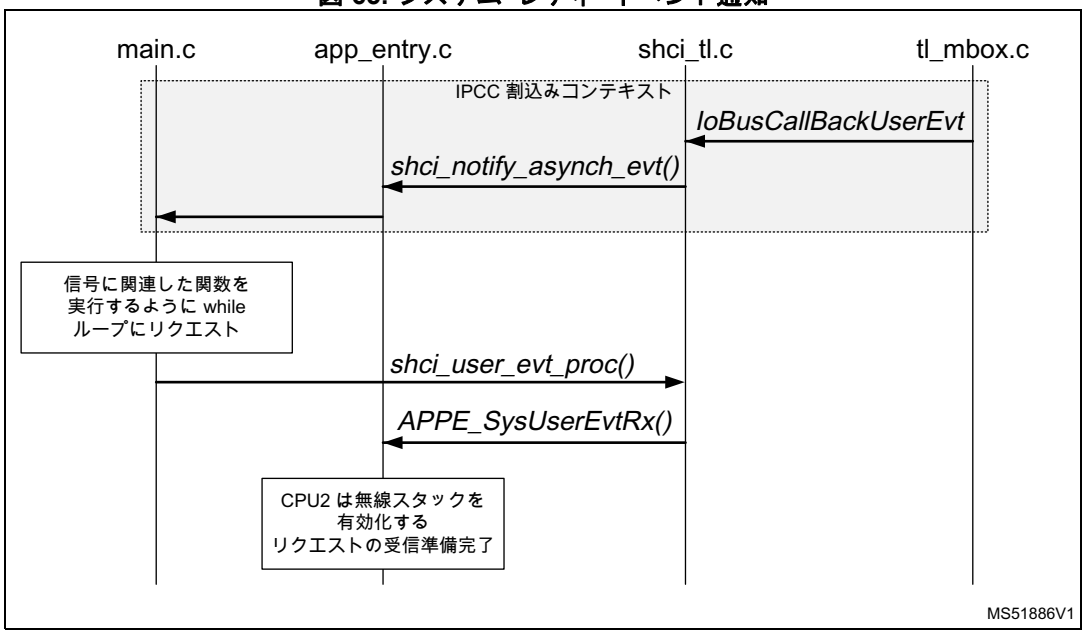
図 64. システムの初期化



CPU2 がシステム·コマンドを受信する準備が整うと、CPU1 に通知を送ります。通知 shci_notify_asynch_evt() を受信したら、shci_user_evt_proc() をコールして、システムトランスポートレイヤがイベントを処理できるようにする必要があります。ユーザアプリケーションは、APPE_SysUserEvtRx() でシステムイベントが受信されたことの通知を受けます。IPCC 割込みハンド

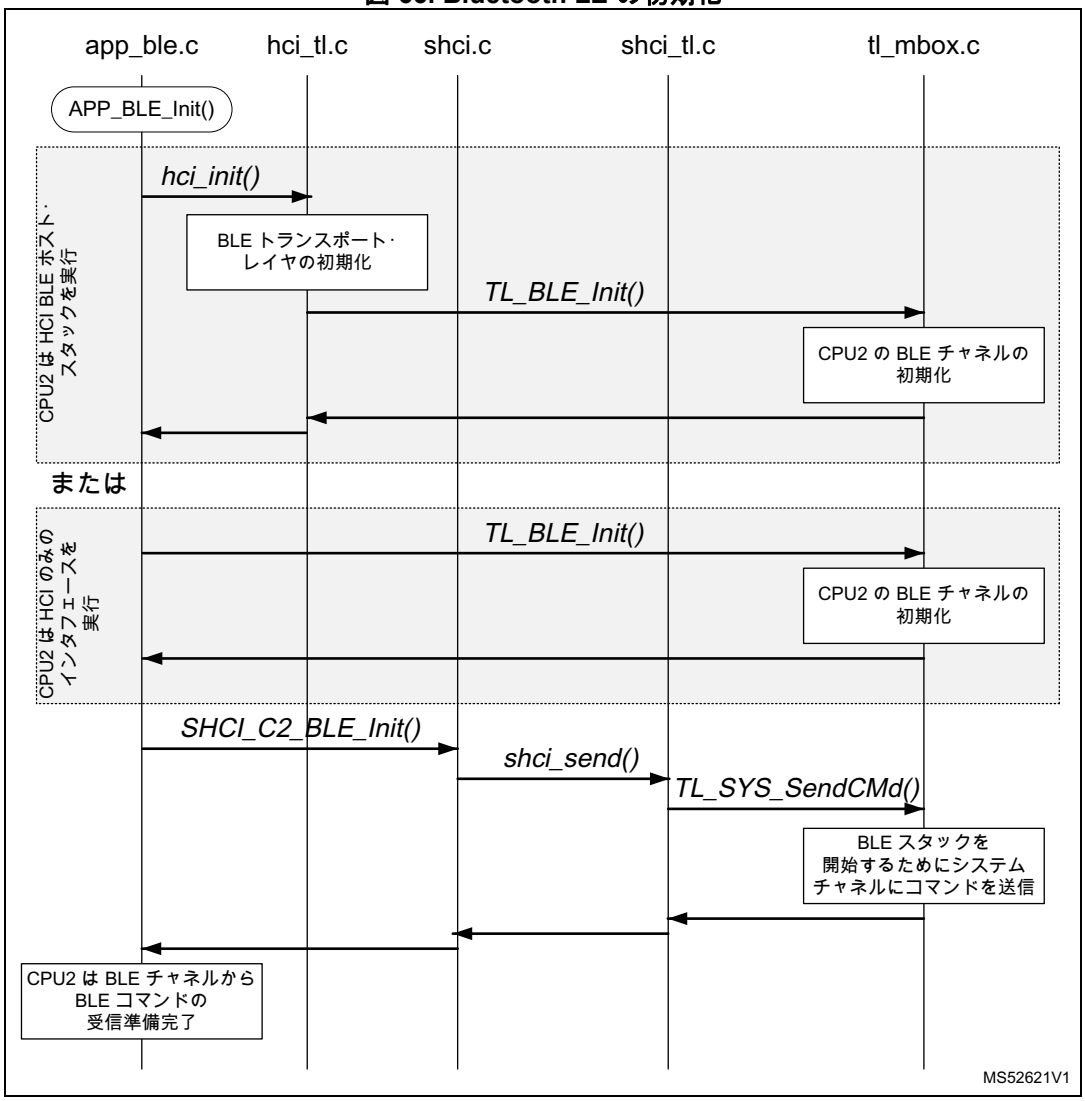
ラコンテキストで shci_notify_asynch_evt() を受信すると、（割込みコンテキスト外の）バックグラウンド while ループから shci_user_evt_proc() がコールされるように、情報はバックグラウンドに渡されます。この仕組みは、CPU2 が Bluetooth LE ホスト･スタック全体、HCI のみのインタフェース、OpenThread スタックのどれを実行していても同じです。

図 65. システム･レディ･イベント通知



参考資料

図 66. Bluetooth LE の初期化



システム・イベントを受信すると、Bluetooth LE トランスポートレイヤが初期化され、CPU2にシステム・コマンドが送信されて Bluetooth LE スタックが開始されます。SHCI_C2_BLE_Init() システム・コマンドが CPU2 に送信されるとすぐに、CPUは Bluetooth LE コマンドを受信する準備が整います。

CPU2 が HCI のみインタフェースを実行している場合、Bluetooth LE トランスポートレイヤは CPU1 の Host Bluetooth LE スタックの中で動作を開始します。したがって、提供されている Bluetooth LE トランスポートレイヤは使用することも初期化することもできません。

13.2 メールボックス·インタフェース

このインタフェースは、Bluetooth LE コントローラへのコマンドの送信に使用しなければならない最低レベルのものです。これはトランスペアレント・モードのアプリケーションで使用されますが、BT SIG HCI インタフェースの上に Bluetooth LE スタック·オープンソースが使用される場合には使用しないでください。

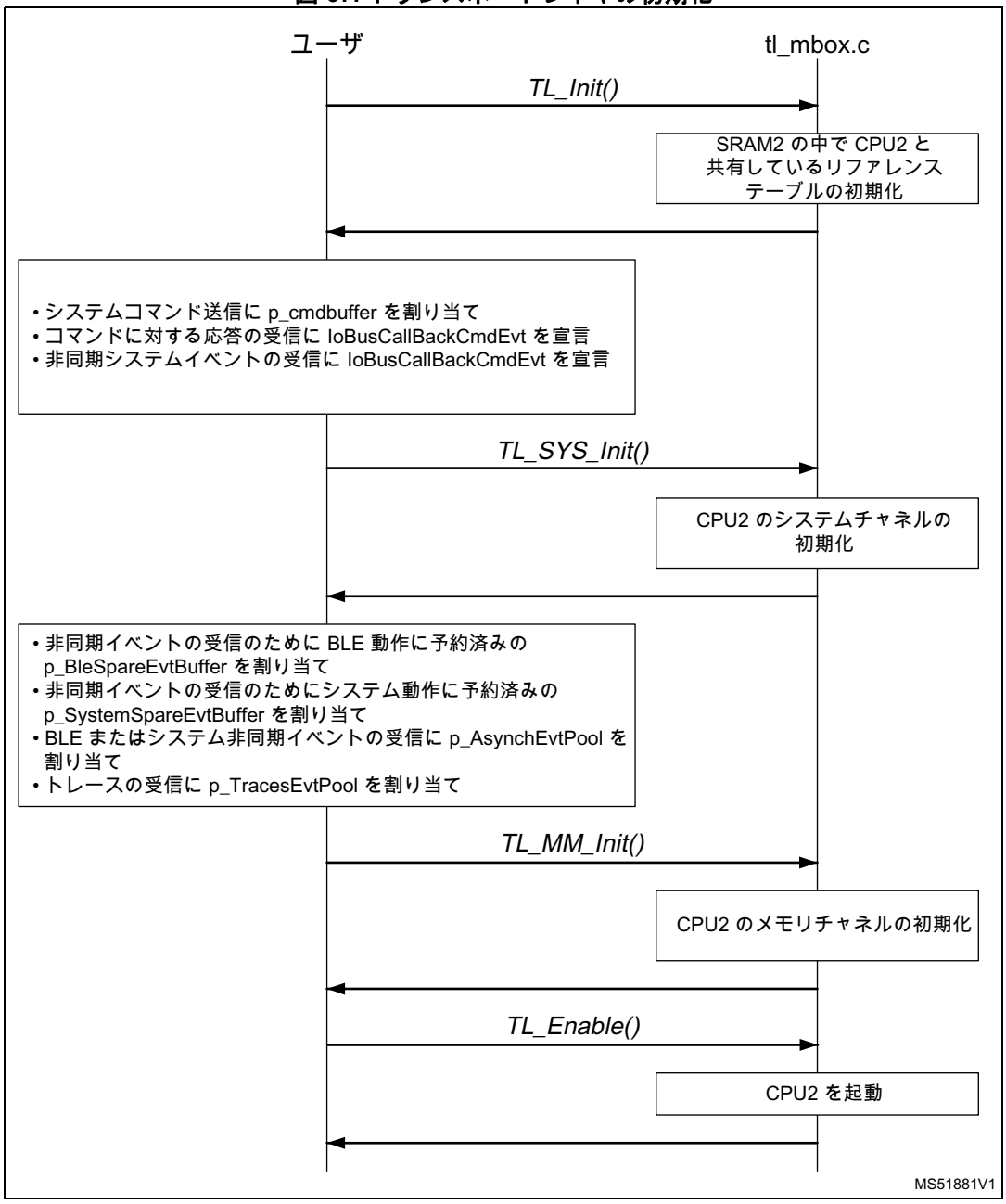
13.2.1 インタフェース API

表 30. インタフェース API

機能	説明
`void TL_Init( void )`	共有メモリを初期化します。
`void TL_Enable( void )`	トランスポートレイヤを有効化します。
`int32_t TL_SYS_Init( void* pConf )`	システムチャネルを初期化します。
`int32_t TL_SYS_SendCmd( uint8_t* buffer, uint16_t size )`	システムコマンドを送信します。
`int32_t TL_BLE_Init( void* pConf )`	Bluetooth LE チャネルを初期化します。
`int32_t TL_BLE_SendCmduint8_t* buffer, uint16_t size )`	Bluetooth LE コマンドを送信します。
`int32_t TL_BLE_SendAclData( uint8_t* buffer, uint16_t size )`	ACL データ·パケットを送信します。
`void TL_MM_Init( TL_MM_Config_t *p_Config )`	メモリチャネルを初期化します。
`void TL_MM_EvtDone( TL_EvtPacket_t * hcievt )`	メモリチャネルにバッファを開放します。

13.2.2 インタフェースと動作の詳細

図 67. トランスポートレイヤの初期化



`void TL_Init( void );`

これは、送信する最初のコマンドです。メールボックス·ドライバと共有メモリを初期化します。

`int32_t TL_SYS_Init( void* pConf );`

ユーザは最初に、メールボックス·ドライバがシステムコマンドの送信に使用するバッファ（p_cmdbuffer）、システムコマンド応答の受信に使用される 2つのコールバック（IoBusCallBackCmdEvt）、システム非同期イベント（IoBusCallBackUserEvt）を割り当てる必要があります。

IoBusCallBackCmdEvt には、新しいシステムコマンドが送信されるのは、前回のものへの応答が受信済みである場合のみとする新しい要件を実装します。

このコマンドは、メールボックス·ドライバのシステムチャネルを初期化します。

```
void TL_MM_Init( TL_MM_Config_t *p_Config ):
```

最初に、メールボックス·ドライバが Bluetooth LE 非同期イベントの報告のみに使用するバッファ（p_BleSpareEvtBuffer）、メールボックス·ドライバがシステム非同期イベントの報告のみに使用するバッファ（p_SystemSpareEvtBuffer）、Bluetooth LE コントローラが Bluetooth LE 非同期イベントとシステム非同期イベントのいずれかを報告するために使用するメモリのプール（p_AsynchEvtPool）、CPU2がトレースの報告に使用するメモリのプール（p_TracesEvtPool）を割り当てる必要があります。

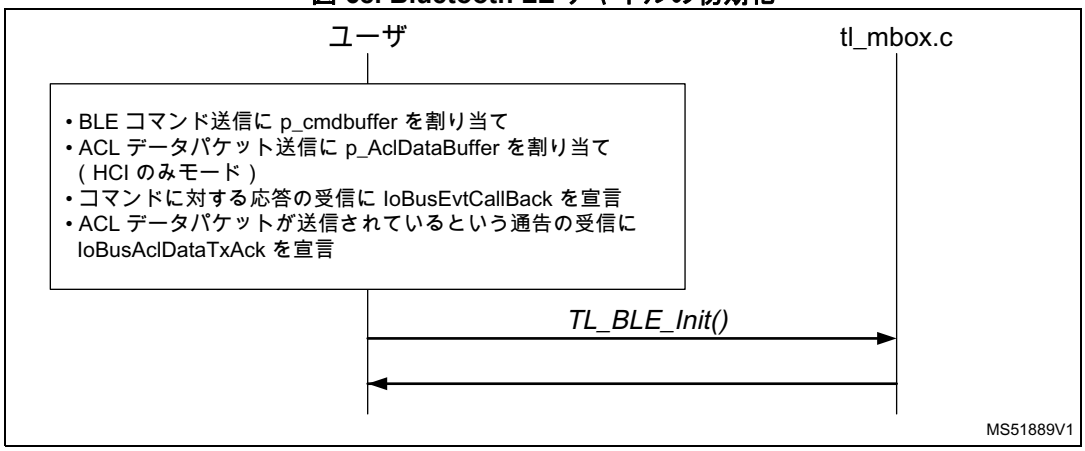
p_BleSpareEvtBuffer バッファと p_SystemSpareEvtBuffer バッファは、たとえメモリ·プール p_AsynchEvtPool が空であっても、CPU2は常に Bluetooth LE イベントとシステムイベントのいずれかを報告可能であることを保証するために使用されます。

このコマンドは、メールボックス·ドライバのメモリ·チャネルを初期化します。

```
void TL_Enable( void ):
```

メールボックス·ドライバが完全に初期化されると、このコマンドが送信されて CPU2 を起動します。

図 68. Bluetooth LE チャネルの初期化




```
int32_t TL_BLE_Init( void* pConf ):
```

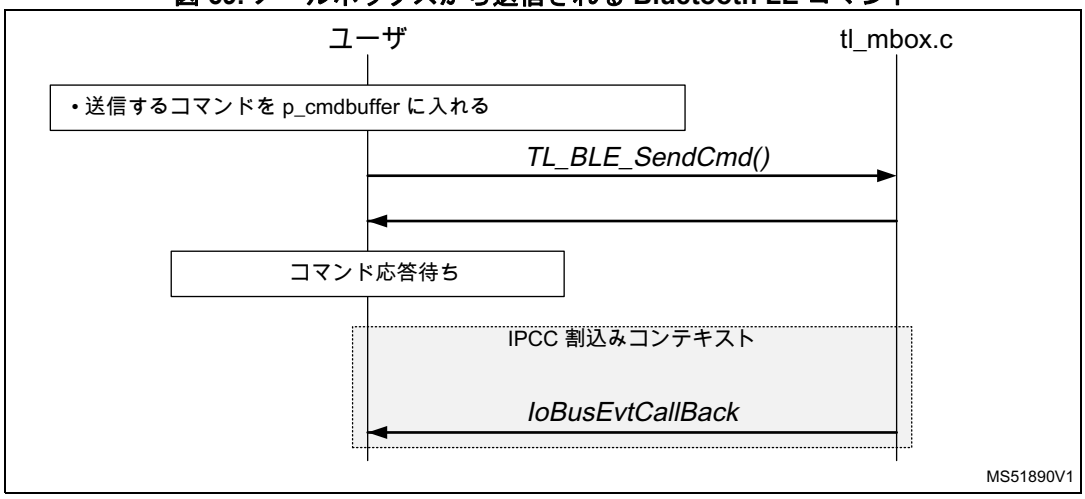
最初に、メールボックス·ドライバが Bluetooth LE コマンドの送信に使用するバッファ（p_cmdbuffer）、メールボックス·ドライバが ACL データ·パケットの送信に使用するバッファ（p_AclDataBuffer）、Bluetooth LE イベントの受信に使用される 2つのコールバック（IoBusEvtCallBack）、ACL データ·パケットの確認応答（IoBusAclDataTxAck）を割り当てる必要があります。

IoBusEvtCallBack を使用して、新しい Bluetooth LE コマンドを送信できるのは（BT SIG によって規定の）コマンド·フローがそれを許可している場合のみであるという要件に適合する必要があります。

HCI のみモードではない場合、p_AclDataBuffer と IoBusAclDataTxAck は両方とも使用されず、0 にセットされる必要があります。

このコマンドによって Bluetooth LE コントローラが初期化されます。

図 69. メールボックスから送信される Bluetooth LE コマンド

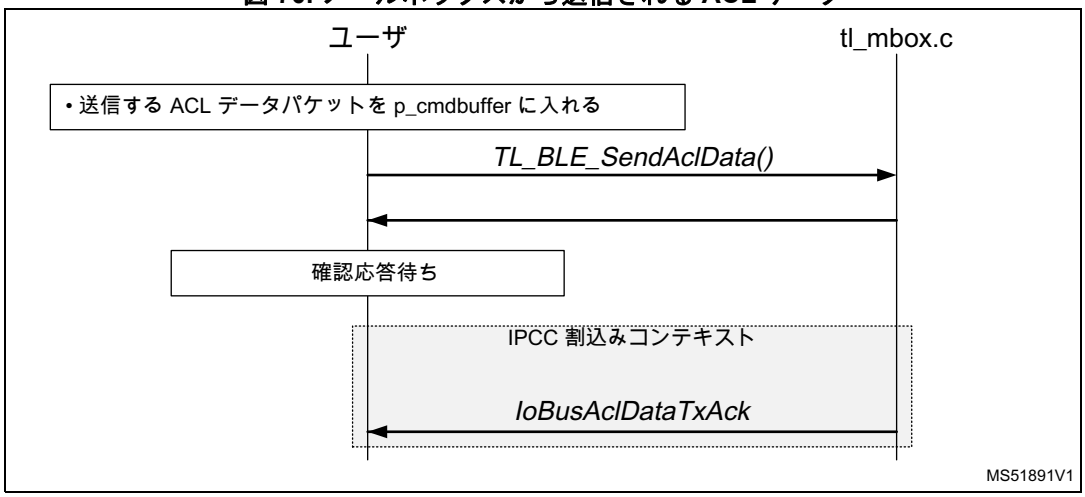



```
int32_t TL_BLE_SendCmd( uint8_t* buffer, uint16_t size ):
```

最初に、送信するコマンドでバッファ p_cmdbuffer をフィルする必要があります。パラメータ·バッファとパラメータ·サイズは使用しません。

IoBusEvtCallBack で受信するコマンド応答を待って応答パケット内のフロー·コマンド制御を確認し、新しいコマンドの送信が可能であるか否かを理解する必要があります。IoBusEvtCallBack は IPCC 割込みコンテキストの中で非同期に生成されます。処理負荷次第では、バックグラウンド·メカニズムを実装して、（IPCC 割込みコンテキストの外で）受信パケットをデコードすることを推奨します。

図 70. メールボックスから送信される ACL データ




```
int32_t TL_BLE_SendAclData( uint8_t* buffer, uint16_t size ):
```

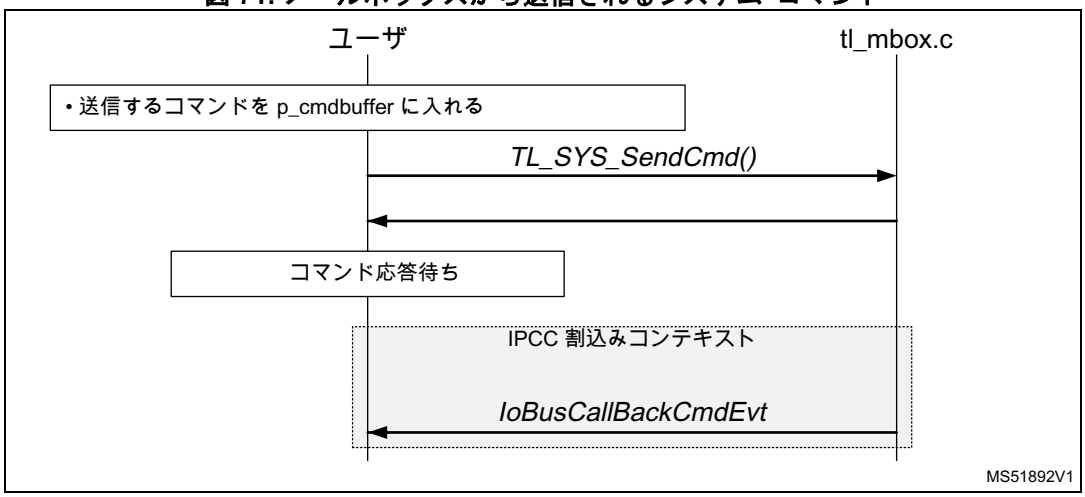
最初に、送信するデータパケット ACL データ·パケットで p_AclDataBuffe バッファを埋める必要があります。パラメータ·バッファとパラメータ·サイズは使用しません。

新しい ACL データパケットを送信する前に、IoBusAclDataTxAck で受信する確認応答を待つ必要があります。IoBusAclDataTxAck は IPCC 割込みコンテキストの中で非同期に生成されます。処理負荷

次第では、バックグラウンド·メカニズムを実装して、(IPCC 割込みコンテキストの外で) 確認応答を処理することを推奨します。

ACL データ·パケット·インタフェースは、HCI モードでのみサポートされます。サポートされている場合、Bluetooth LE コマンドのペンディング中に ACL データ·パケットの送信が可能です。Bluetooth LE コマンドと ACL データ·パケットはリソースを共有しません。

図 71. メールボックスから送信されるシステム·コマンド

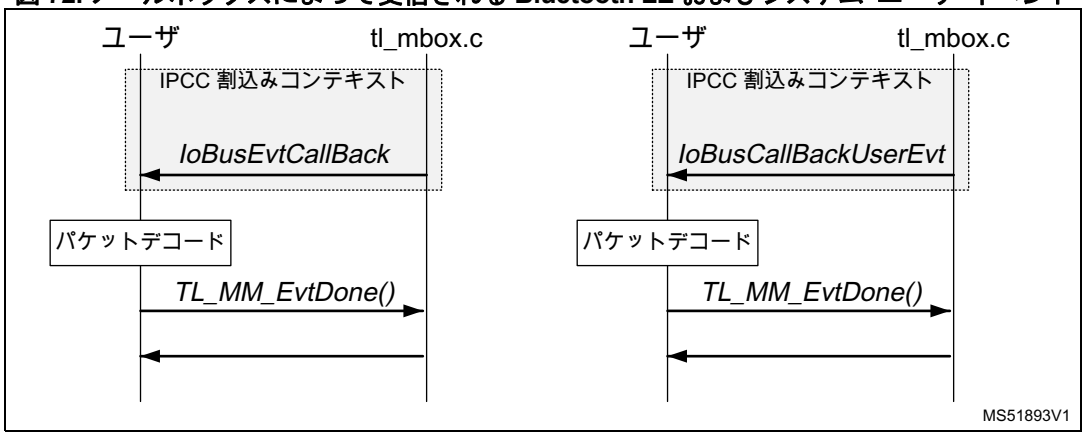



```
int32_t TL_SYS_SendCmd( uint8_t* buffer, uint16_t size )
```

最初に、送信するコマンドで p_cmdbuffer バッファをフィルする必要があります。パラメータ·バッファとパラメータ·サイズは使用しません。

新しいコマンドを送信する前に、IoBusCallBackCmdEvt で受信するコマンド応答を待つ必要があります。IoBusCallBackCmdEvt は IPCC 割込みコンテキストの中で非同期に生成されます。処理負荷次第では、バックグラウンド·メカニズムを実装して、(IPCC 割込みコンテキストの外で) 受信パケットをデコードすることを推奨します。

図 72. メールボックスによって受信される Bluetooth LE およびシステム·ユーザ·イベント




```
void TL_MM_EvtDone( TL_EvtPacket_t * hcievt ):
```

このAPIを呼び出して、次の場合にCPU2で実行されているメモリマネージャーにパケットを返す必要があります。

- Bluetooth LE コマンド応答ではない IoBusEvtCallBack（ユーザ Bluetooth LE イベント･コールバック）で受信した各パケットに対して
- IoBusCallBackUserEvt（ユーザ･システム･イベント･コールバック）で受信した各パケットに対して

13.3 メールボックス･インタフェース - 拡張

メールボックスによって送信されるバッファの中に送信するコマンドをビルドする場合には、メールボックス･インタフェースが適しています。同様に、受信したイベントパケットをデコードし、コマンドフロー制御を管理して新しいコマンドが送信可能か確認する必要があります。

これは、HCI インタフェースの上に CPU1 で動作する Bluetooth LE ホストスタックを使用する場合です。この場合、CPU2 は HCI のみモードで使用されます。

ただし、Bluetooth LE ホスト･スタックは、CPU2 の初期化に必要なシステム･チャネルをサポートしていません。したがって、メールボックス･インタフェースのみを使用する場合には、CPU2 に送信するシステムコマンドパケットをビルドして、CPU2 から受信したイベントを管理する必要があります。

システム･メールボックス･インタフェースに接続するときに、シンプル Bluetooth LE メールボックス･インタフェースと上位レベルの shci インタフェースを混合して、システムパケットのエンコード／デコードを行うことが可能です。これが「メールボックス･インタフェース - 拡張」の目的です。

13.3.1 インタフェース API

Bluetooth LE インタフェースとメモリ･インタフェースは、シンプル･メールボックス･インタフェースと同じものです。

（shci.h から）上位の shci インタフェースを使用するには、shci トランスポートレイヤを初期化してメールボックス･ドライバに接続する必要があります。

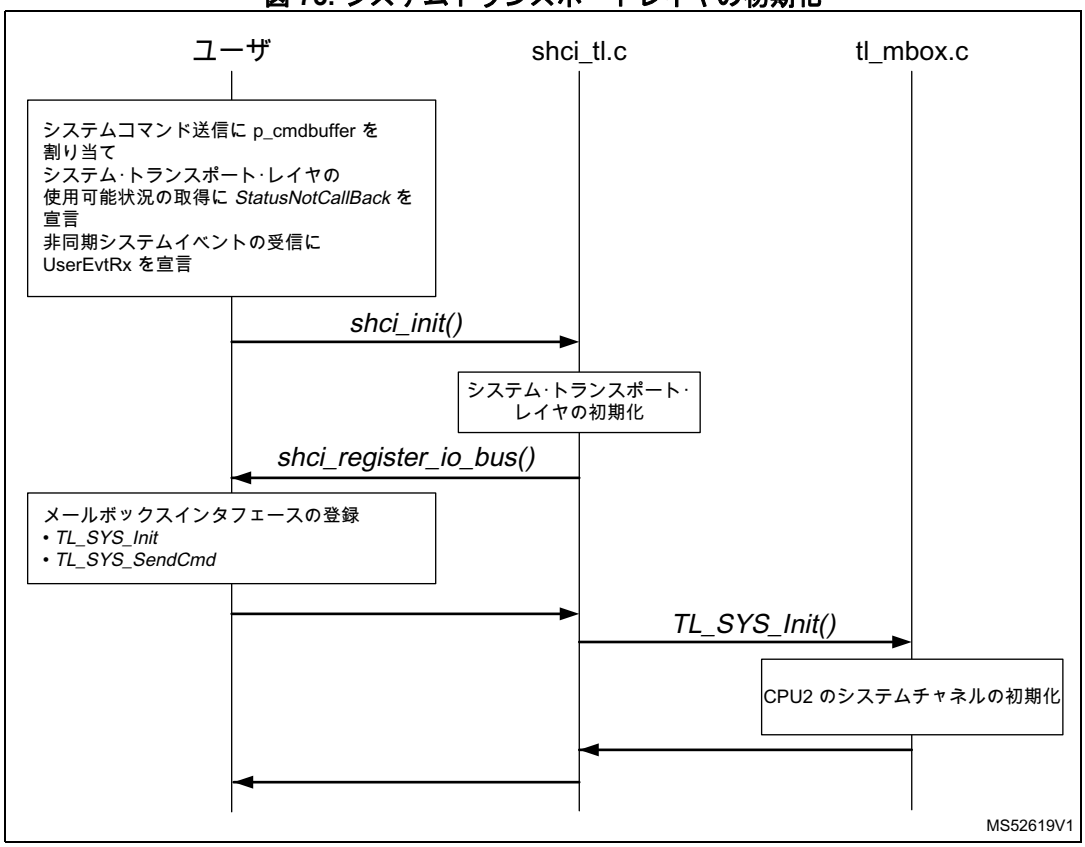
API TL_SYS_Init() および TL_SYS_SendCmd() の 2つと、2つのコールバック IoBusCallBackCmdEvt および IoBusCallBackUserEvt は、トランスポートレイヤで使用と実装が行われ、それ以上個別には使用できなくなります。

表 31. インタフェース API

機能	説明
`void shci_init(void(* UserEvtRx)(void* pData), void* pConf)`	システム･トランスポート･レイヤを初期化します。
`void shci_register_io_bus(tSHciIO* fops)`	システム・トランスポート・レイヤにメールボックス･インタフェースを登録します。
`void shci_notify_asynch_evt(void* pdata)`	shci_user_evt_procを呼び出すようにユーザーに要求します。
`void shci_resume_flow(void)`	停止されたのはいつであるかを報告する非同期ユーザ･イベントを再開します。
`void shci_cmd_resp_wait(uint32_t timeout)`	コマンド応答を待ちます。
`void shci_cmd_resp_release(uint32_t flag)`	コマンド応答を受信したことを通知します。
`void shci_user_evt_proc(void)`	受信した非同期ユーザイベントを処理し、UserEvtRx をコールします。

13.3.2 インタフェースと動作の詳細

図 73. システムトランスポートレイヤの初期化




```
void shci_init(void(* UserEvtRx)(void* pData), void* pConf):
```

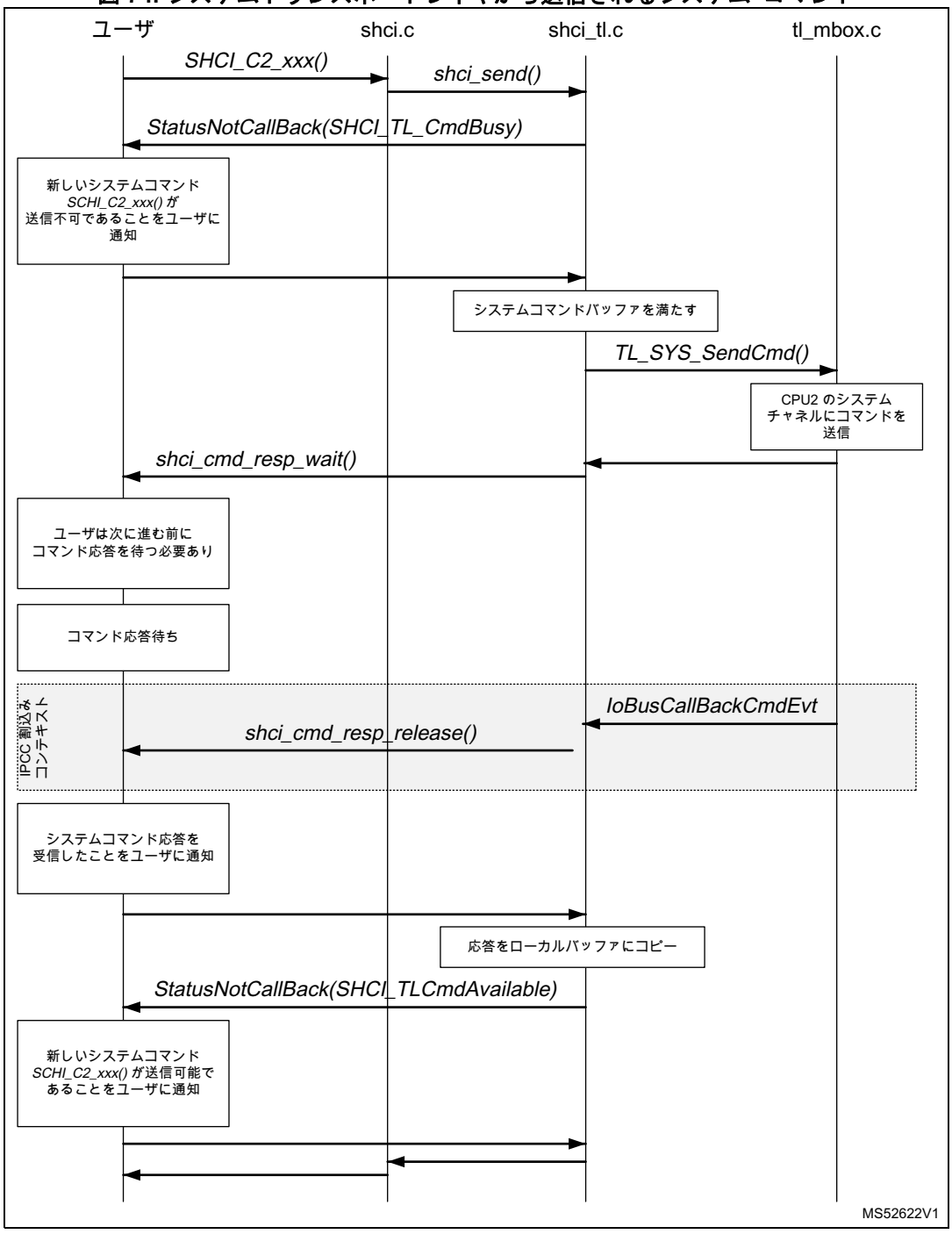
ユーザは最初に、メールボックス·ドライバがシステムコマンドの送信に使用するバッファ（p_cmdbuffer）、ユーザ非同期システムイベントを受信する 2つのコールバック（UserEvtRx）、トランスポートレイヤ使用可能通知（StatusNotCallBack）を割り当てる必要があります。

このコマンドは、トランスポートレイヤとメールボックス·ドライバのシステムチャネルを初期化します。

```
void shci_register_io_bus(tSHciIO* fops);:
```

このコマンドは、システムトランスポートレイヤにメールボックス·ドライバを登録します。

図 74. システムトランスポートレイヤから送信されるシステム･コマンド



SHCI_C2_xxx()

アプリケーションが使用可能なサポートされているシステムコマンドのリストは、ファイル shci.h にあります。

`void StatusNotCallBack(SHCI_TL_CmdStatus_t status):`

これは、システムコマンドが送信可能である場合に確認応答を行う shci_init() に登録されているコールバックです。異なるスレッドからシステムコマンドが送信される可能性があるマルチスレッド･アプリケーションで使用する必要があります。

status = SHCI_TL_CmdBusy である場合、システムトランスポートレイヤはビジーであり、新しいシステムコマンドの送信はできません。

`void shci_cmd_resp_wait(uint32_t timeout):`

アプリケーションは、応答が受信済みであることを通知する shci_cmd_resp_wait() と必ず一緒にこのコマンドを使用する必要があります。

パラメータは意味を持ちません。

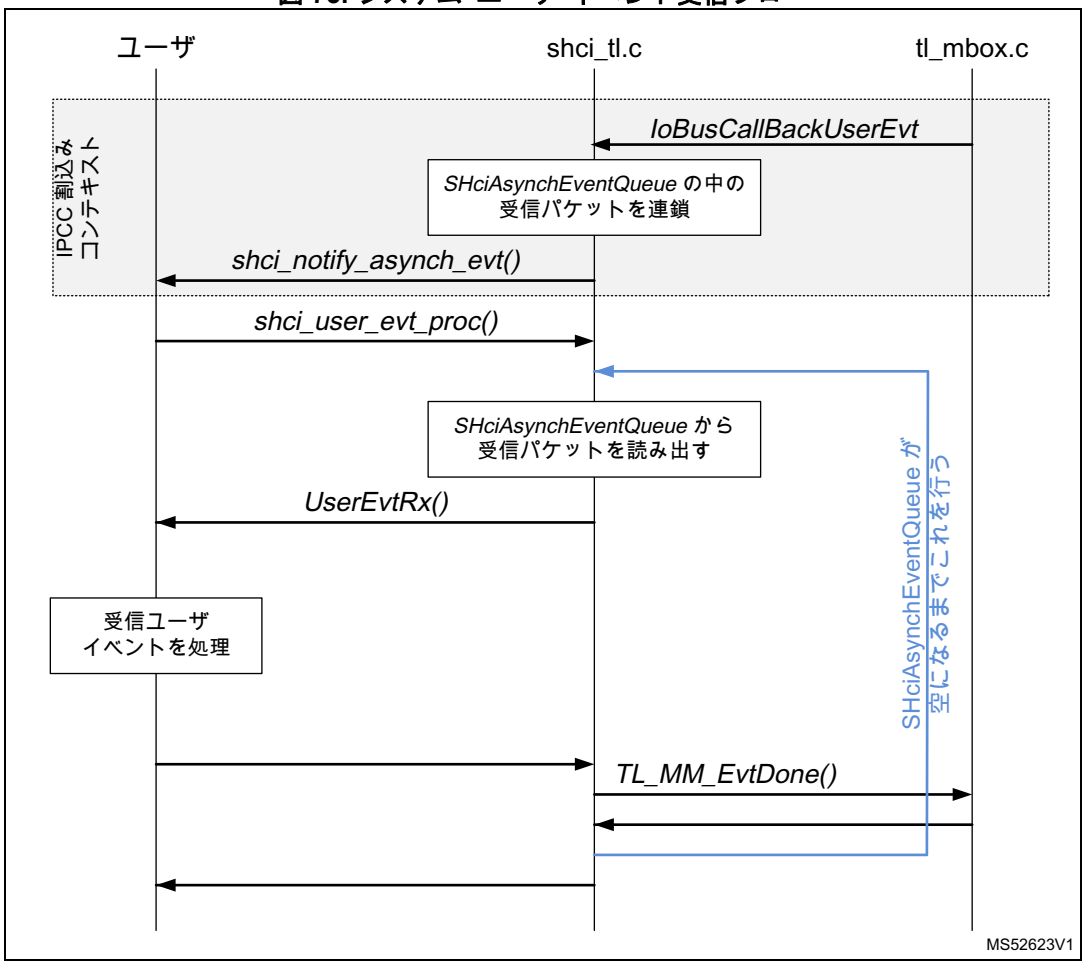
`void shci_cmd_resp_wait(uint32_t timeout):`

この関数は、ペンディング中のシステムコマンドの応答が受信されたことを通知します。

これは、IPCC 割込みコンテキストと呼ばれます。この API から抜けるときに、アプリケーションは API shci_cmd_resp_wait() から戻ることができます。

パラメータは意味を持ちません。

図 75. システム･ユーザ･イベント受信フロー



`void shci_notify_asynch_evt(void* pdata):`

この API は、システム･ユーザ･イベントが受信されたことを通知します。shci_user_evt_proc() をコールして、システムトランスポートレイヤで通知を処理する必要があります。shci_notify_asynch_evt() 通知は IPCC 割込みコンテキストからコールされますので、バックグラウンド･メカニズムを実装して、（IPCC 割込みコンテキストの外で）shci_user_evt_proc() をコールすることを強く推奨します。

pdata には SHciAsynchEventQueue のアドレスが保持されます。

`void shci_user_evt_proc(void):`

この関数は、UserEvtRx() で受信したイベントを報告します。受信イベント･キュー SHciAsynchEventQueue は IPCC 割込みコンテキストの中でフィルされますので、イベントを処理している間に新しいイベントをキューに格納することができます。UserEvtRx() はキューから取得したイベントごとにコールされます。shci_user_evt_proc() プロセスは、UserEvtRx() から戻るたびに CPU2 メモリマネージャにバッファを開放します。

```
void UserEvtRx (void * pData):
```

この関数は、受信したシステムイベントを報告します。受信イベントを格納するバッファは、この関数からの戻りで開放されます。

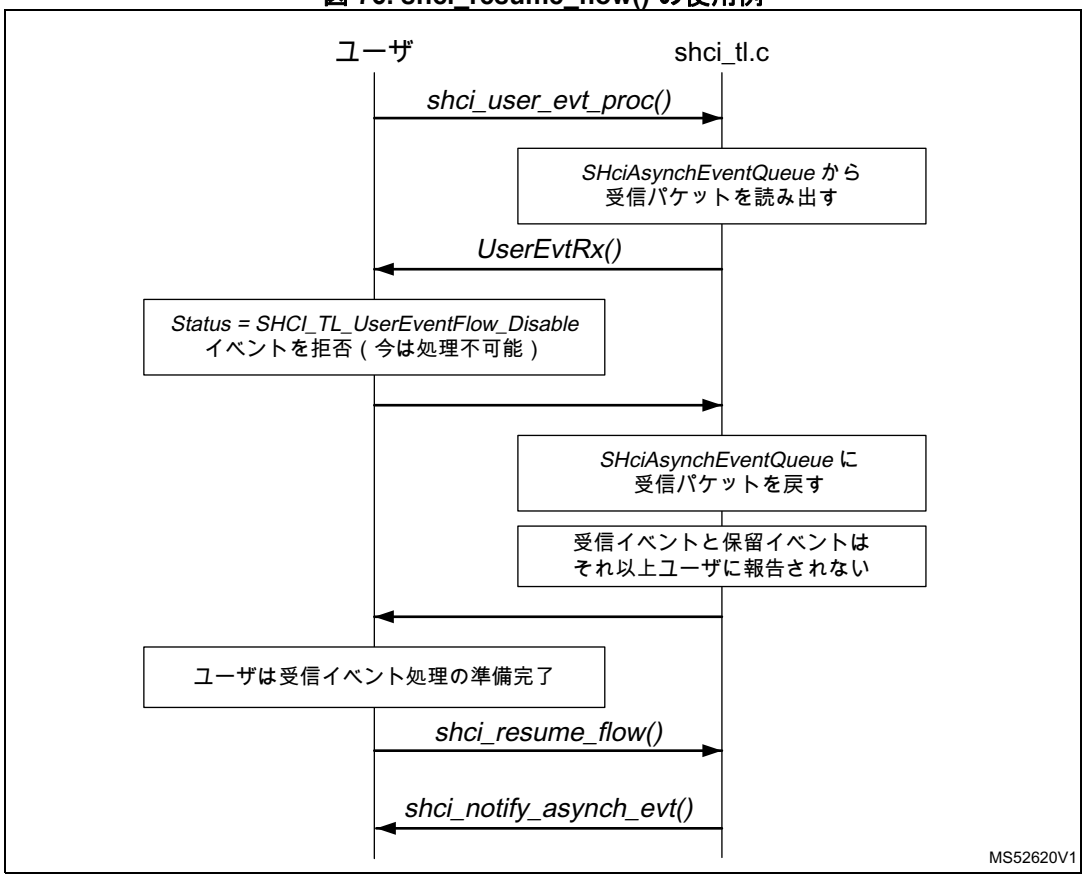
pData は、次のパラメータを保持する構造体のアドレスです。

```
typedef struct
{
SHCI_TL_UserEventFlowStatus_t status;
TL_EvtPacket_t *pckt;
} tSHCI_UserEvtRxParam;
```

pckt：受信イベントのアドレスが保持されます。

status：受信パケットが未処理であり、破棄してはならないことをシステムトランスポートレイヤに通知する方法を提供します。UserEvtRx() からの戻りでフィルされない場合、このパラメータは SHCI_TL_UserEventFlow_Enable にセットされますが、これは受信イベントが処理済みであることを意味しています。

図 76. shci_resume_flow() の使用例




```
void shci_resume_flow( void ):
```

受信イベントを処理できない場合には、UserEvtRx() から戻る前にステータス·パラメータを SHCI_TL_UserEventFlow_Disable にセットする必要があります。その場合、システム・トランスポート・レイヤはシステムイベントを開放せず、新しい受信イベントを報告しません。

システムイベントを処理する準備が整っている場合、システムイベントの報告を再開するようにシステム・トランスポート・レイヤに通知する shci_resume_flow() を送信する必要があります。

13.4 ACI インタフェース

これは、CPU2上で動作する Bluetooth LE スタックに対するインタフェースです。Bluetooth LE レイヤ（GATT、GAP、HCI LE）のすべての機能を使用するための API のフルセットを提供します。

ACI コマンドは HCI トランスポート全体に送信されます。

すべての Bluetooth LE レイヤ（GATT、GAP）にアクセスするためのインタフェースは、フォルダ \Middlewares\ST\STM32_WPAN\ble\core\Inc\core にあります。

ACI インタフェースを使用するときには、Bluetooth LE コントローラをフルスタックモードにセットする必要があります。ACI インタフェースからメールボックスにコマンドを送受信するには、HCI トランスポートレイヤをアプリケーションに実装する必要があります。

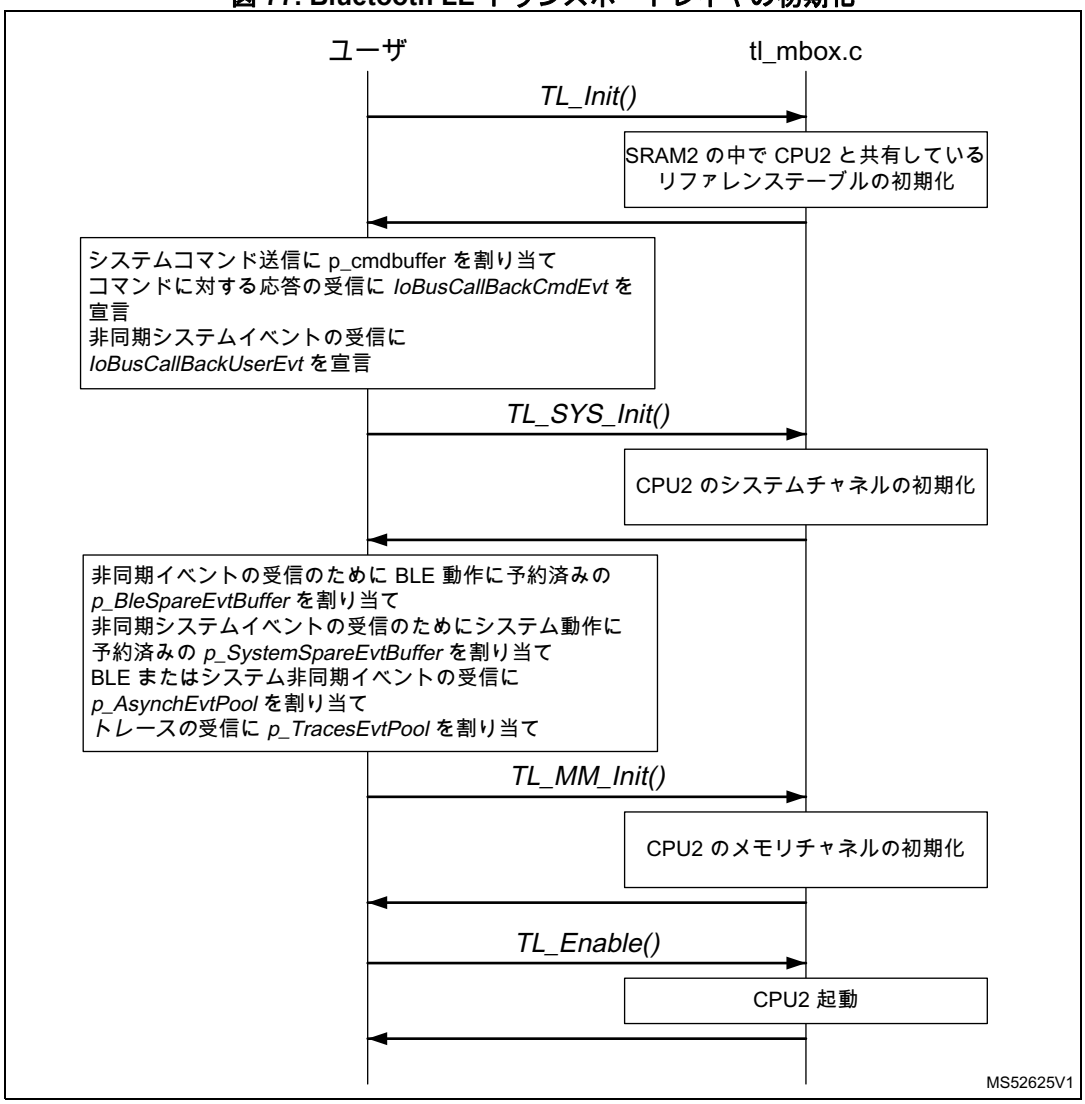
新しいインタフェースは、基本的にメールボックス·インタフェース - 拡張であり、HCI トランスポートレイヤが実装されています。ACI インタフェースを使用する場合、アプリケーションはローレベル・メールボックス·インタフェースを使用 しません。

表 32. Bluetooth LE トランスポートレイヤのインタフェース

機能	説明
`void hci_init(void(* UserEvtRx)(void* pData), void* pConf);`	Bluetooth LE トランスポート·レイヤを初期化します。
`void hci_register_io_bus(tHciIO* fops);`	Bluetooth LE トランスポートレイヤにメールボックス·インタフェースを登録します。
`void hci_notify_asynch_evt(void* pdata);`	ユーザに hci_user_evt_proc のコールをリクエストします。
`void hci_resume_flow (void)`	停止されたのはいつであるかを報告する非同期ユーザ·イベントを再開します。
`void hci_cmd_resp_wait(uint32_t timeout)`	コマンド応答を待ちます。
`void hci_cmd_resp_release(uint32_t flag)`	コマンド応答を受信したことを通知します。
`void hci_user_evt_proc(void`	受信した非同期ユーザイベントを処理し、UserEvtRx をコールします。

13.4.1 インタフェースと動作の詳細

図 77. Bluetooth LE トランスポートレイヤの初期化




```
void hci_init(void(* UserEvtRx)(void* pData), void* pConf);:
```

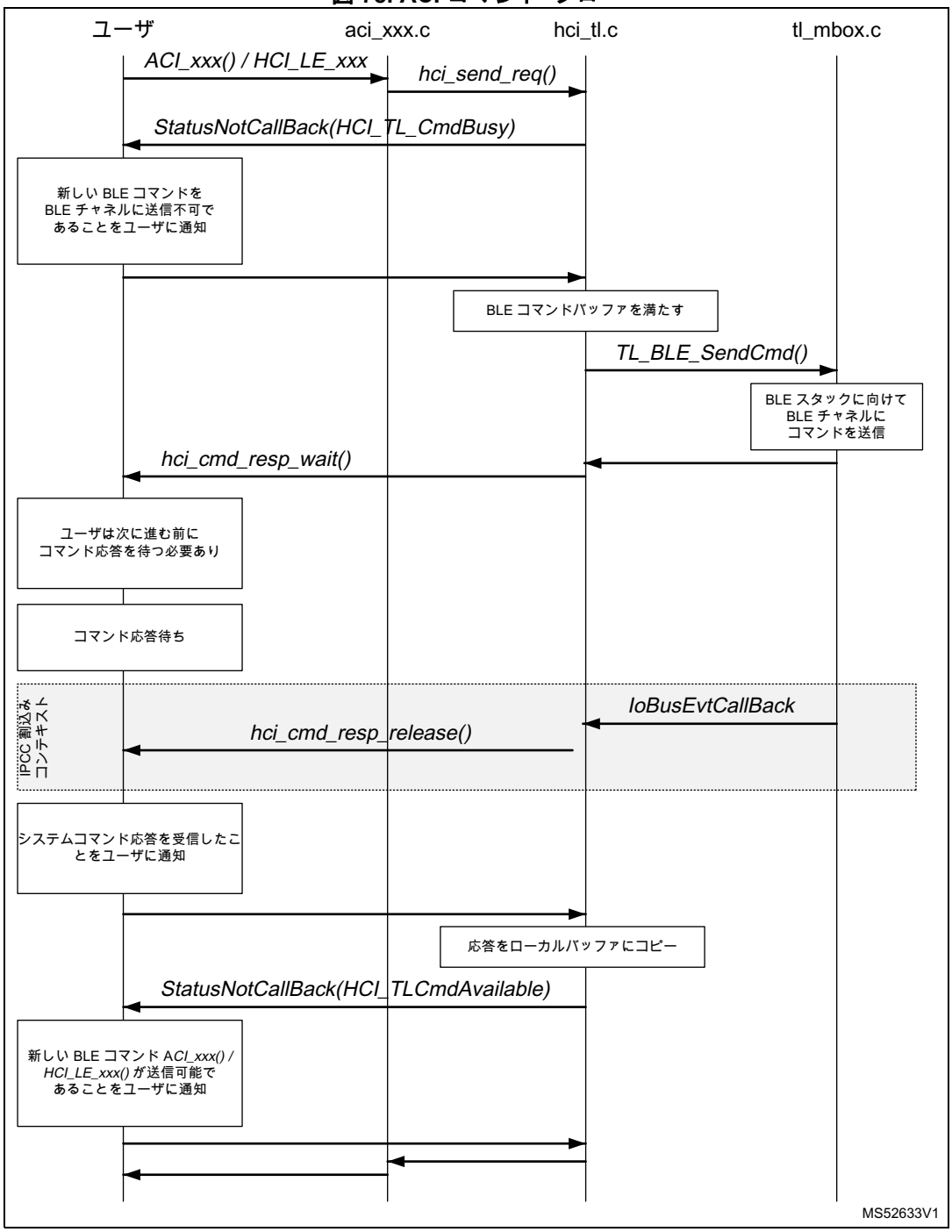
最初に、メールボックス·ドライバが Bluetooth LE コマンドの送信に使用するバッファ（p_cmdbuffer）、ユーザ非同期システムイベントの受信に使用される 2つのコールバック（UserEvtRx）、トランスポートレイヤ使用可能通知（StatusNotCallBack）を割り当てる必要があります。

このコマンドは、HCI トランスポートレイヤとメールボックス·ドライバの Bluetooth LEチャネルを初期化します。

```
void hci_register_io_bus(tSHciIO* fops);:
```

このコマンドは、HCI トランスポートレイヤにメールボックス·ドライバを登録します。

図 78. ACI コマンド・フロー



ACI_xxx() / HCI_LE_xxx()

アプリケーションが使用可能なサポートされているシステムコマンドのリストは、フォルダ \Middlewares\ST\STM32_WPAN\ble\core\Inc\core にあります。

`void StatusNotCallBack(HCI_TL_CmdStatus_t status):`

これは、Bluetooth LE コマンドが送信可能である場合に確認応答を行う hci_init() に登録されているコールバックです。Bluetooth LE コマンドを別スレッドから送信できるマルチスレッド･アプリケーションの中で使用するためのものです。

status = HCI_TL_CmdBusy である場合､HCI トランスポートレイヤはビジーであり､新しい Bluetooth LE コマンドの送信はできません。

`void hci_cmd_resp_wait(uint32_t timeout):`

アプリケーションは、応答が受信されたことを通知するために hci_cmd_resp_wait() がコールされるまで、このコマンドから戻ることができません。

パラメータは意味を持ちません。

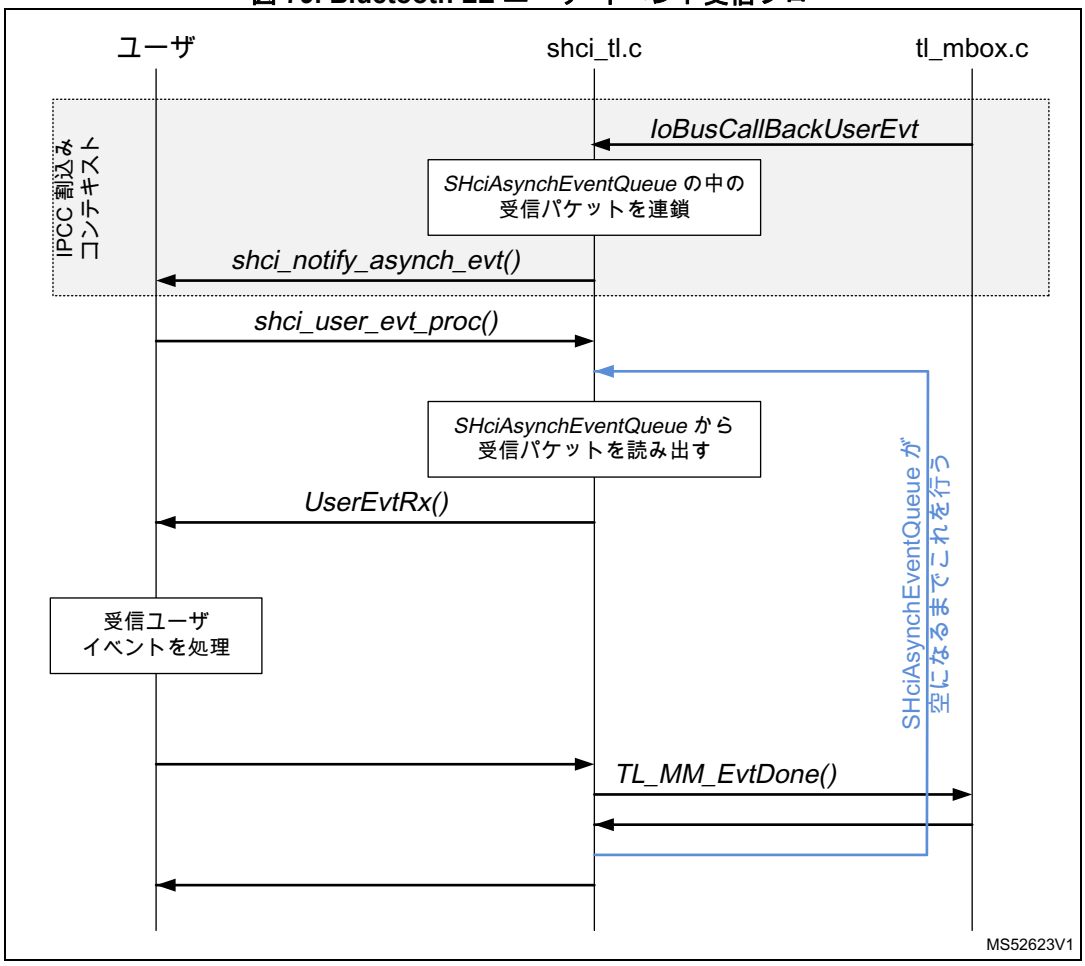
`void hci_cmd_resp_wait(uint32_t timeout):`

この関数は、ペンディング中の Bluetooth LE コマンドの応答が受信されたことを通知します。

これは、IPCC 割込みコンテキストと呼ばれます。この API から抜けるときに、アプリケーションは API hci_cmd_resp_wait() から戻ることができます。

パラメータは意味を持ちません。

図 79. Bluetooth LE ユーザ･イベント受信フロー



`void hci_notify_asynch_evt(void* pdata):`

この API は、Bluetooth LE ユーザ･イベントが受信されたことを通知します。その後に、hci_user_evt_proc() をコールして、HCI トランスポートレイヤで通知を処理する必要があります。hci_notify_asynch_evt() 通知は IPCC 割込みコンテキストからコールされますので、バックグラウンド･メカニズムを実装して、（IPCC 割込みコンテキストの外で）hci_user_evt_proc() をコールすることを強く推奨します

pdata は HciCmdEventQueue のアドレスを保持します。

`void hci_user_evt_proc(void):`

この関数は、UserEvtRx() 経由で受信したイベントを報告します。受信イベント･キュー HciCmdEventQueue は IPCC 割込みコンテキストの中で満たされますので、イベントを処理している間に新しいイベントをキューに格納することができます。UserEvtRx() はキューから取得したイベントごとにコールされます。hci_user_evt_proc() プロセスは、それぞれのUserEvtRx()から戻るたびに CPU2 メモリマネージャにバッファを開放することを担当しています。

`void UserEvtRx (void * pData):`

この関数は、受信した Bluetooth LE ユーザイベントを報告します。受信イベントを格納するバッファは、この関数からの戻りで開放されます。

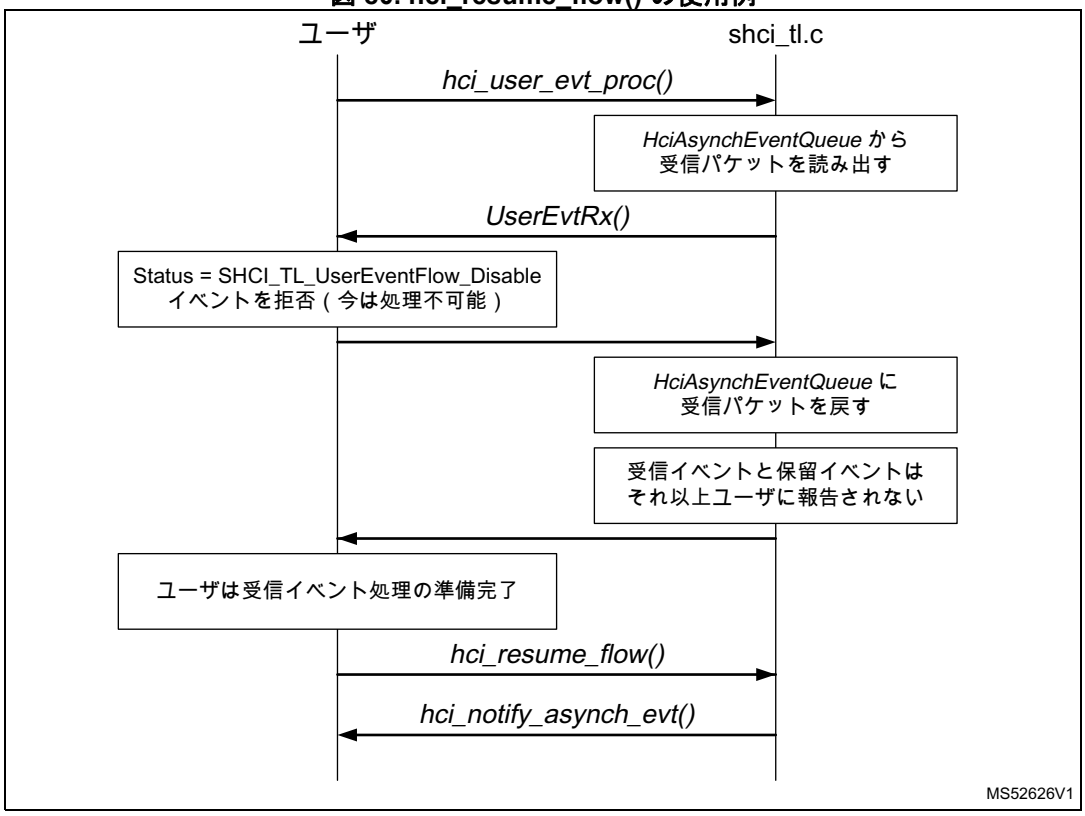
pData は、次のパラメータを保持する構造体のアドレスです。

```
typedef struct
{
HCI_TL_UserEventFlowStatus_t status;
TL_EvtPacket_t *pckt;
} tHCI_UserEvtRxParam;
```

pckt：受信イベントのアドレスが保持されます。

status：受信パケットが未処理であり、破棄してはならないこと をHCI トランスポートレイヤに通知する方法を提供します。UserEvtRx() からの戻りで満たされない場合、このパラメータは HCI_TL_UserEventFlow_Enable にセットされますが、これは受信イベントが処理済みであることを意味しています。

図 80. hci_resume_flow() の使用例




```
void hci_resume_flow( void ):
```

受信イベントを処理できない場合には、UserEvtRx() から戻る前にステータス･パラメータを HCI_TL_UserEventFlow_Disable にセットする必要があります。その場合、HCI トランスポートレイヤは Bluetooth LE ユーザ･イベントを開放せず、新しい受信イベントを報告しません。

Bluetooth LE ユーザ･イベントを処理する準備が整っている場合、Bluetooth LE ユーザ･イベントの報告を再開するように HCI トランスポートレイヤに通知する hci_resume_flow() を送信する必要があります。

13.5 STM32WB システム･コマンドとイベント

13.5.1 コマンド

表 33. システム･インタフェース･コマンド

コマンド	コード	説明
SHCI_C2_FUS_GetState()	0xFC52	[5] を参照してください。
SHCI_C2_FUS_FwUpgrade()	0xFC54	
SHCI_C2_FUS_FwDelete()	0xFC55	
SHCI_C2_FUS_UpdateAuthKey()	0xFC56	
SHCI_C2_FUS_LockAuthKey()	0xFC57	
SHCI_C2_FUS_StoreUsrKey()	0xFC58	
SHCI_C2_FUS_LoadUsrKey()	0xFC59	
SHCI_C2_FUS_StartWs()	0xFC5A	
SHCI_C2_FUS_LockUsrKey()	0xFC5D	
SHCI_C2_BLE_Init()	0xFC66	Bluetooth LE 初期化パラメータを送信します。ACI コマンドの前に送信する必要があります。詳細については、セクション 7.6.5 : データ･スループットを最大化する方法を参照してください。
SHCI_C2_THREAD_Init()	0xFC67	セクション 9.8:Thread･アプリケーションのためのシステム･コマンド を参照してください。
SHCI_C2_DEBUG_Init	0xFC68	CPU1 と CPU2 の両方でトレースを、CPU2 で GPIO デバッグ設定を有効化します。
SHCI_C2_FLASH_EraseActivity	0xFC69	Flash メモリ消去操作が CPU1 からリクエストされていることを CPU2 に通知します。これによって、CPU2 は消去操作に対するタイミング保護を有効にすることができます。
SHCI_C2_CONCURRENT_SetMode()	0xFC6A	セクション 9.8:Thread･アプリケーションのためのシステム･コマンド を参照してください。
SHCI_C2_FLASH_StoreData()	0xFC6B	
SHCI_C2_FLASH_EraseData()	0xFC6C	
SHCI_C2_RADIO_AllowLowPower()	0xFC6D	
SHCI_C2_MAC_802_15_4_Init ()	0xFC6E	セクション 12.4.5 : MAC IEEE Std 802.15.4-2011システム を参照してください。
SHCI_C2_Reinit()	0xFC6F	CPU1 上の SEV 命令によって生成されたイベントの受信時に、初期化フェーズを再開するように CPU2 にリクエストします。RF 動作が開始されていないときに SBSFU によって使用されるためのものです。
SHCI_GetWirelessFwInfo()		CPU2 で動作しているワイヤレススタックおよび FUS のバージョンならびにメモリ･フットプリントを返します。
SHCI_C2_ZIGBEE_Init()	0xFC70	CPU2 上の ZigBee® プロトコル･スタックを初期化します。
SHCI_C2_ExtpaConfig()	0xFC72	外部 PA の有効化／無効化ピンの駆動に使用する GPIO とその設定を CPU2 に送信します。
SHCI_C2_SetFlashActivityControl()	0xFC73	Flashメモリ操作からタイミングを保護するために PESD ビットとセマフォ 7のいずれかを使用することを CPU2にリクエストします。このコマンドが送信されない場合、CPU2 は PESD を使用します。

13.5.2 イベント

表 34. ユーザ･システム･イベント

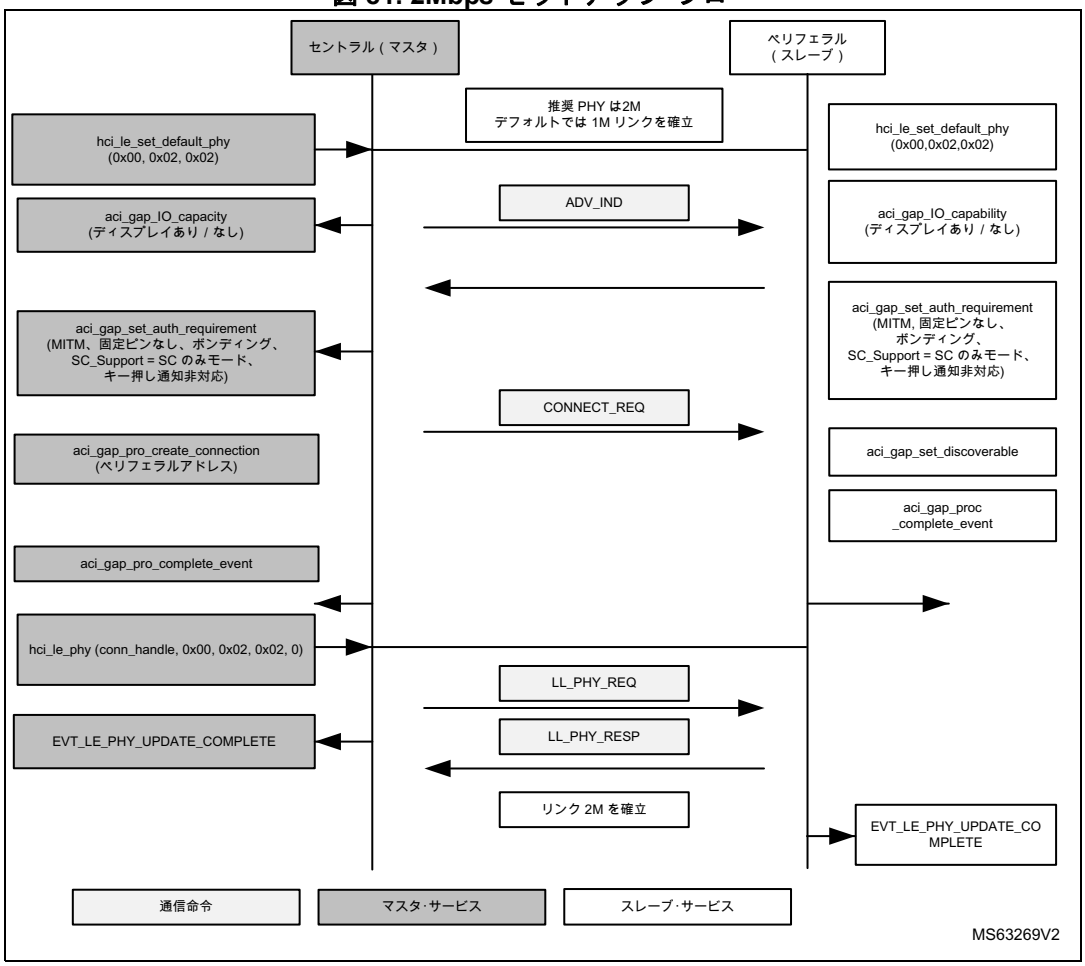
イベント	コード	説明
SHCI_SUB_EVT_CODE_READY	0x9200	CPU2 が起動してコマンドを受け取る準備ができるとすぐに返されます。
SHCI_SUB_EVT_ERROR_NOTIF	0x9201	CPU2からのエラーを報告します。

13.6 Bluetooth LE - 2Mbps リンクの設定

デバイス初期化フェーズにおいて、推奨する TX_PHYS と RX_PHYS の値を初期化できます。

図 81 に示されているように、1Mbps で接続した後、このリンクに対して PHY を 2Mbps に変更することができます。

図 81. 2Mbps セットアップ･フロー

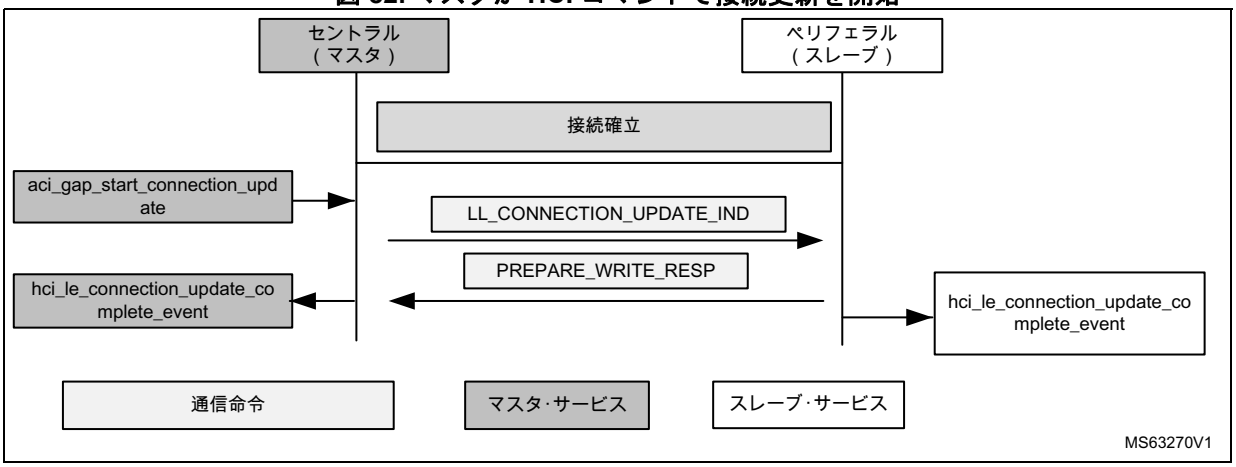


参考資料

13.7 Bluetooth LE - 接続更新手順

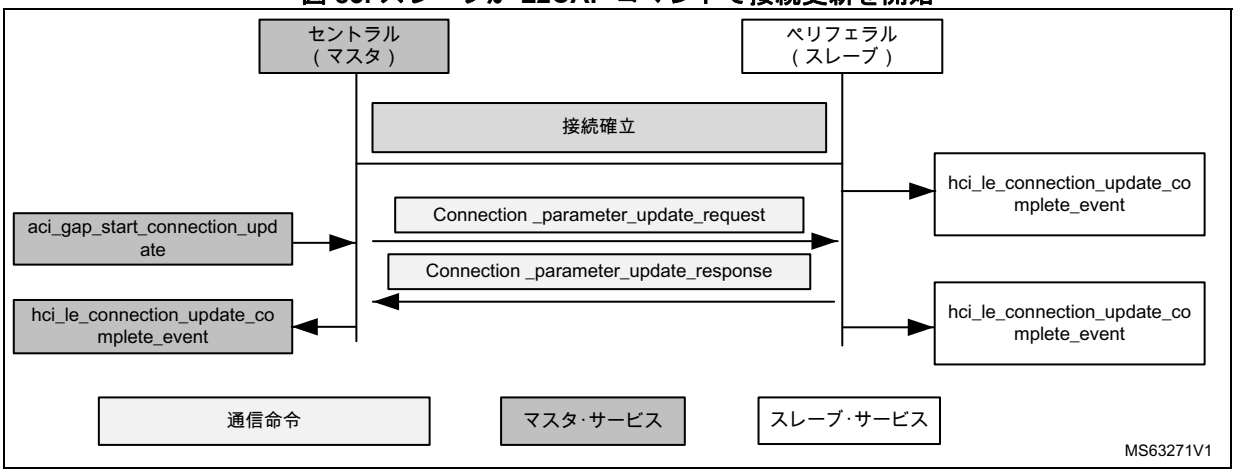
接続が確立されると、マスタは aci_gap_start_connection_update コマンドを用いて接続パラメータの更新が可能となります。

図 82. マスタが HCI コマンドで接続更新を開始



接続が確立されると、スレーブは、aci_l2cap_connection_parameter_update_req コマンドを用いて接続パラメータの更新が可能となります。

図 83. スレーブが L2CAP コマンドで接続更新を開始



13.8 Bluetooth LE - セキュリティ手順

13.8.1 LE セキュリティモード1 レベル 3

図 84、図 85、図 86 に、次の初期化パラメータを使用した LE セキュリティモード 1 レベル 3シーケンスの例を示します。

- セントラルの場合：
  - aci_gap_set_IO_capability（キーボード／ディスプレイ）
  - aci_gap_set_auth_requirement (MITM、固定ピンなし、SC_Support = 0、SC 非サポート)
- ペリフェラルの場合：
  - aci_gap_set_IO_capability（ディスプレイのみ）
  - aci_gap_set_auth_requirement (MITM なし、固定ピン = 0x1234、SC_Support = 0、SC 非サポート)

図 84. LE セキュリティモード 1レベル 3キー配布機能

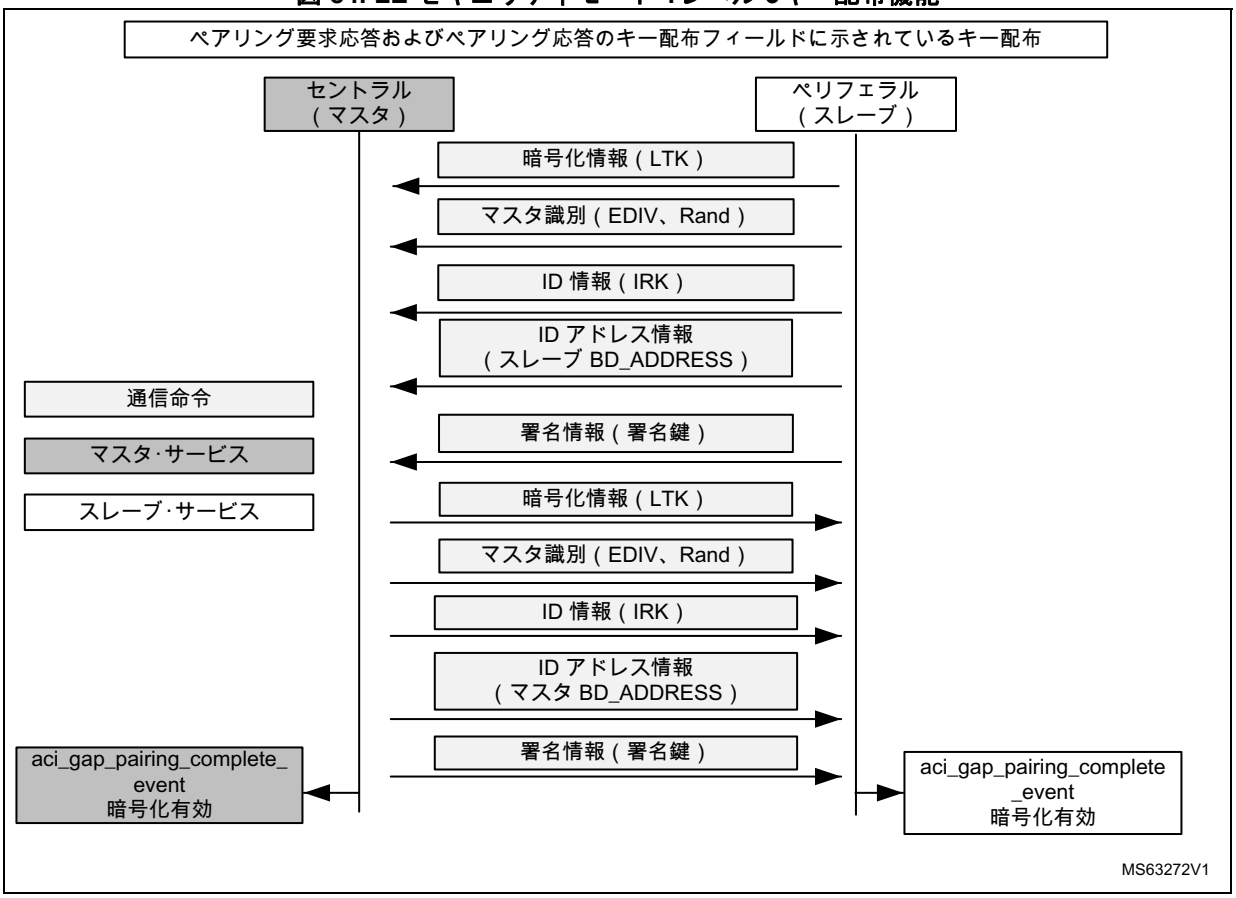


図 85. LE セキュリティモード 1レベル 3ペアリング機能交換

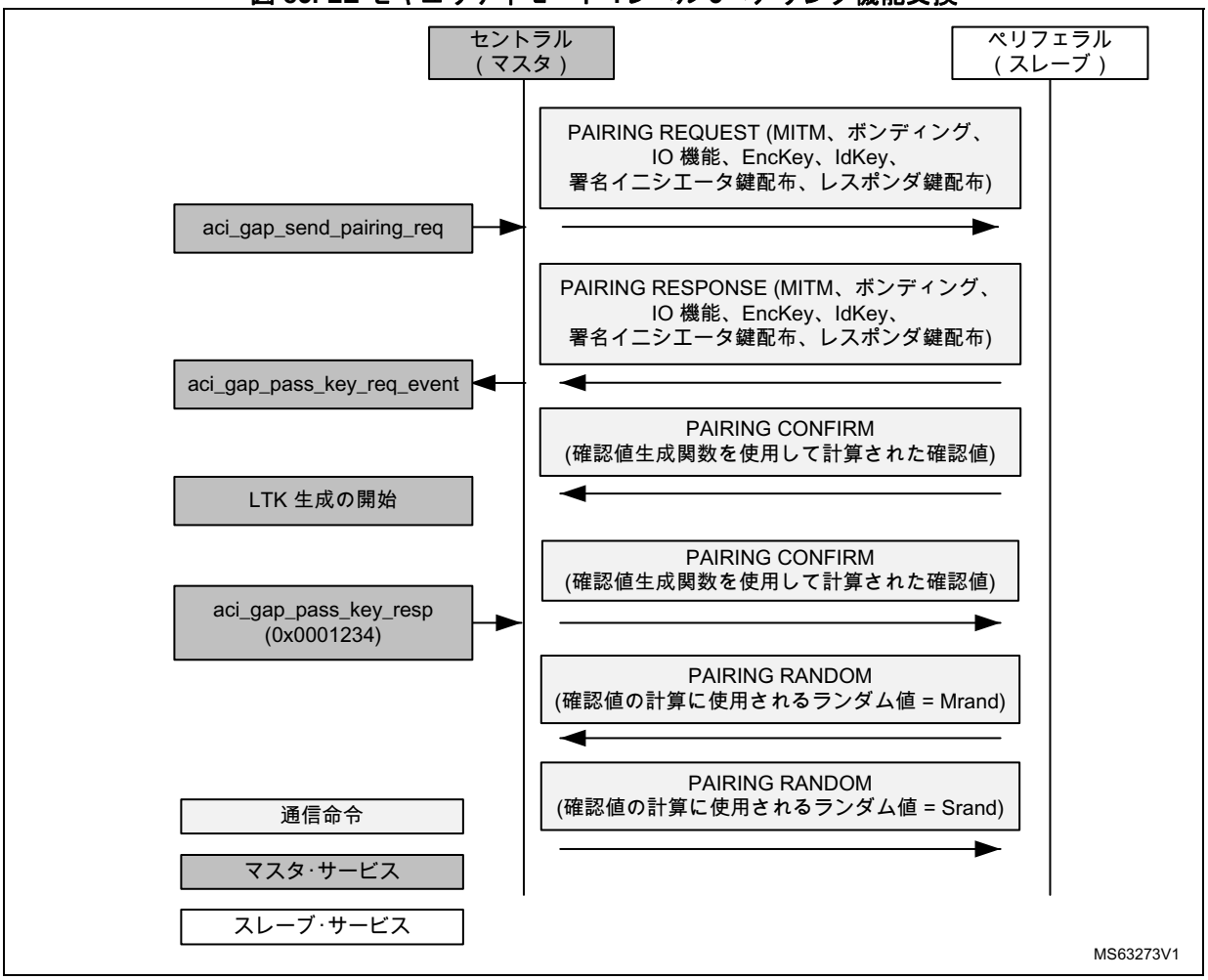
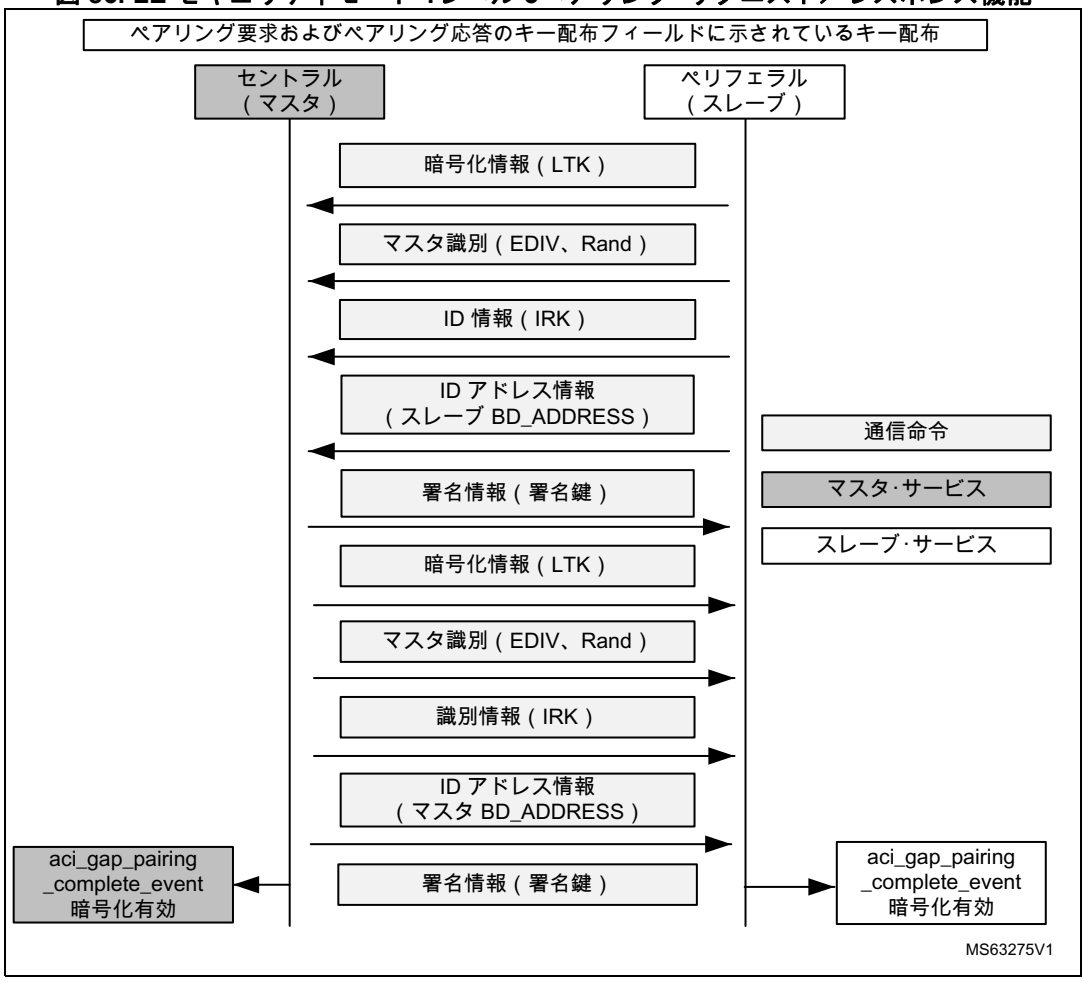


図 86. LE セキュリティモード 1レベル 3ペアリング･リクエスト／レスポンス機能



13.8.2 LE セキュリティモード 1レベル 4

図 87、図 88、図 89 に、次の初期化パラメータを使用したセキュリティモード 1レベル 4シーケンスの例を示します。

- aci_gap_set_IO_capability（display_yesno）
- aci_gap_set_auth_requirement (MITM、固定ピンなし、ボンディング、SC_Support = SC のみモード)

参考資料

図 87. LE セキュリティモード 1レベル 4

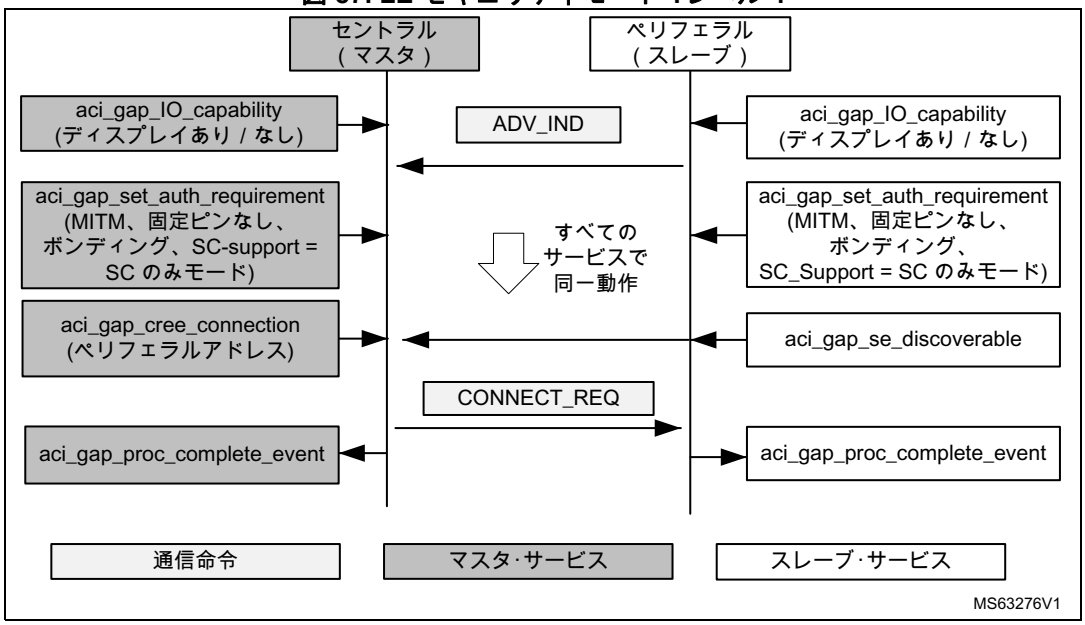


図 88. LE セキュリティモード 1レベル 4

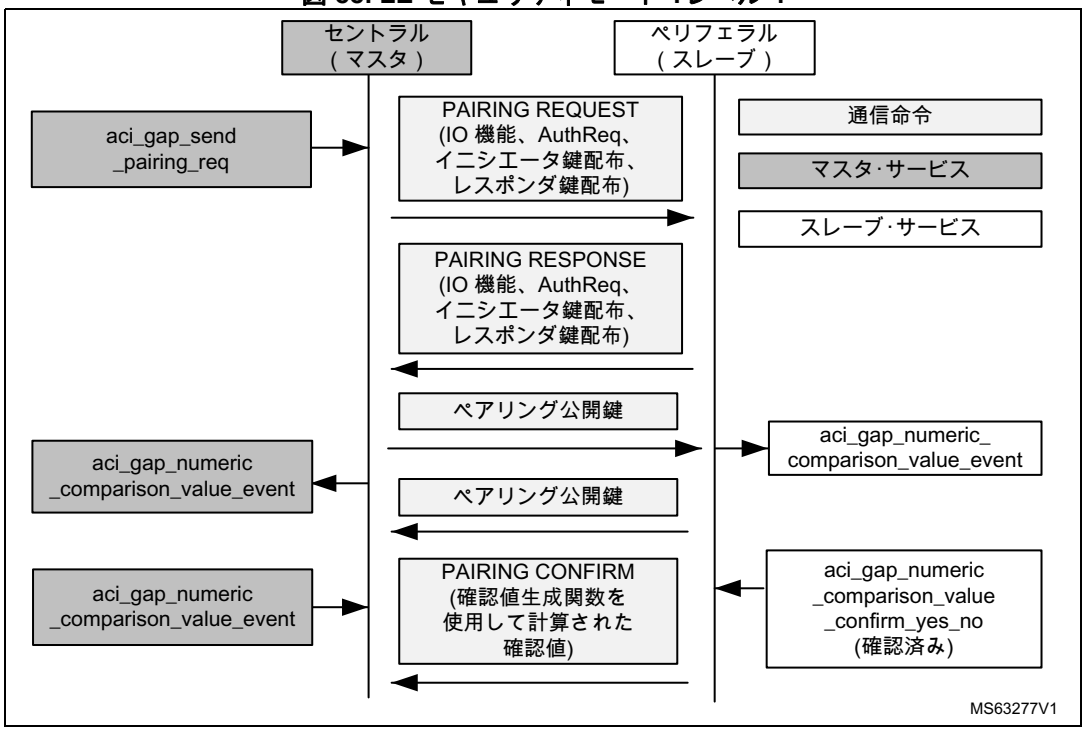
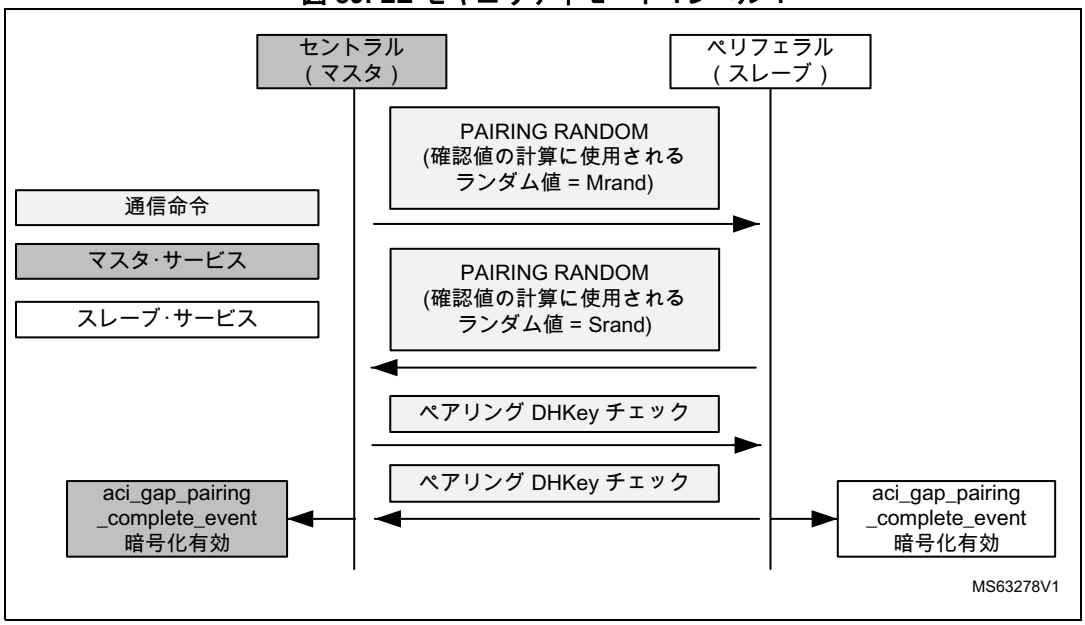


図 89. LE セキュリティモード 1レベル 4



13.8.3 LE セキュリティモード 1レベル 4 - キー押し通知

図 90 と図 91 に、次の初期化パラメータを使用したセキュリティモード 1レベル 4シーケンスの例を示します。

- aci_gap_set_IO_capability（display_yesno）
- aci_gap_set_auth_requirement (MITM、固定ピンなし、SC_Support = SC のみモード、キー押し通知対応)

参考資料

図 90. LE セキュリティモード 1レベル 4 - キー押し通知（1/2）

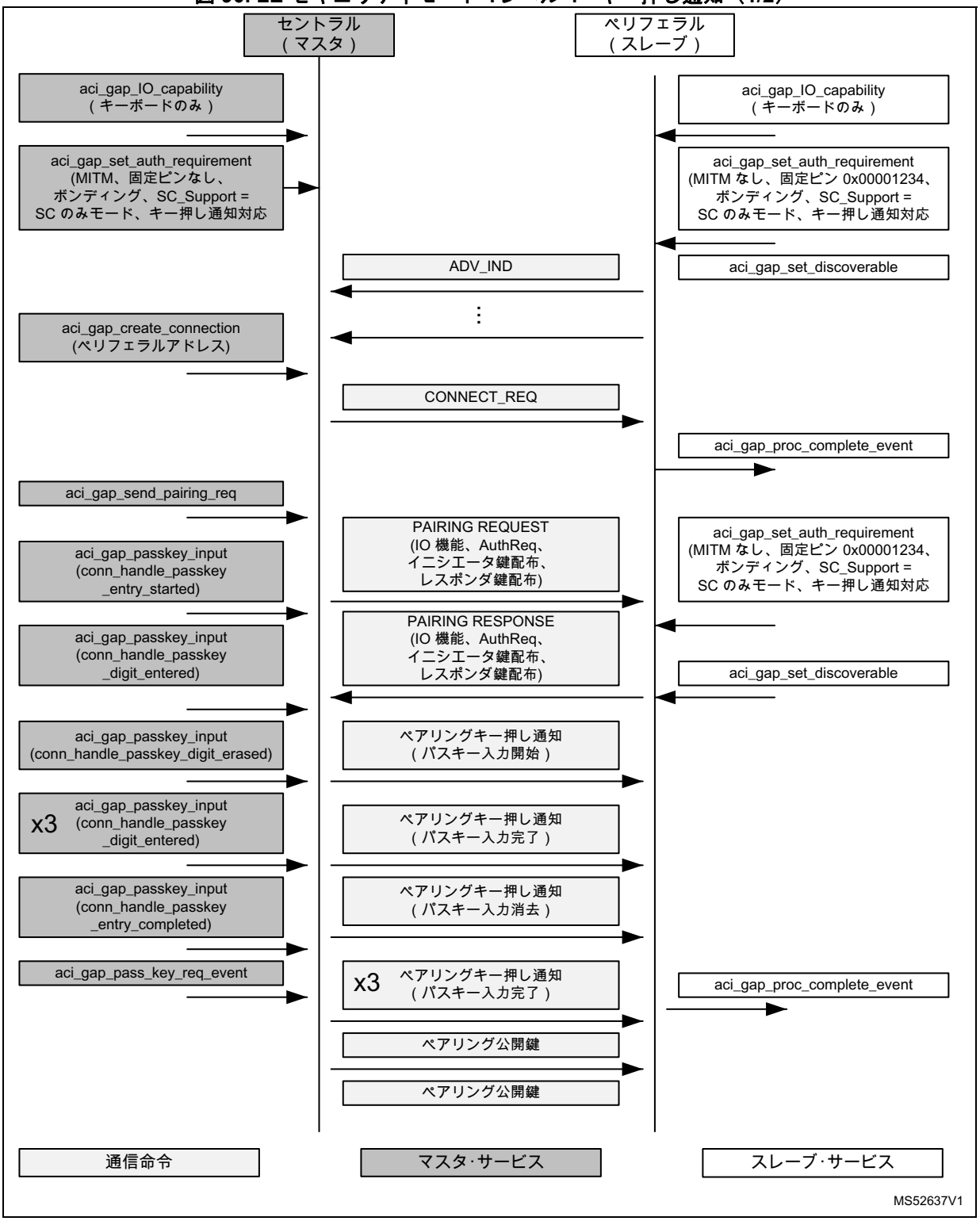
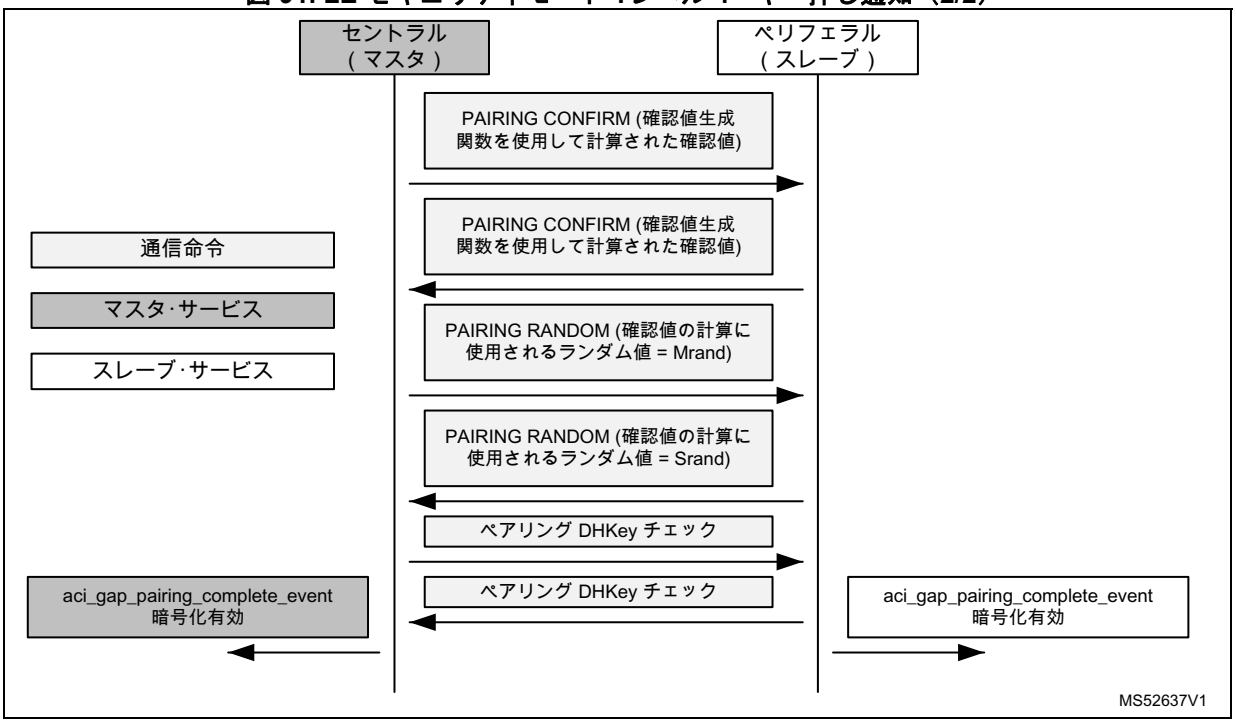


図 91. LE セキュリティモード 1レベル 4 - キー押し通知（2/2）

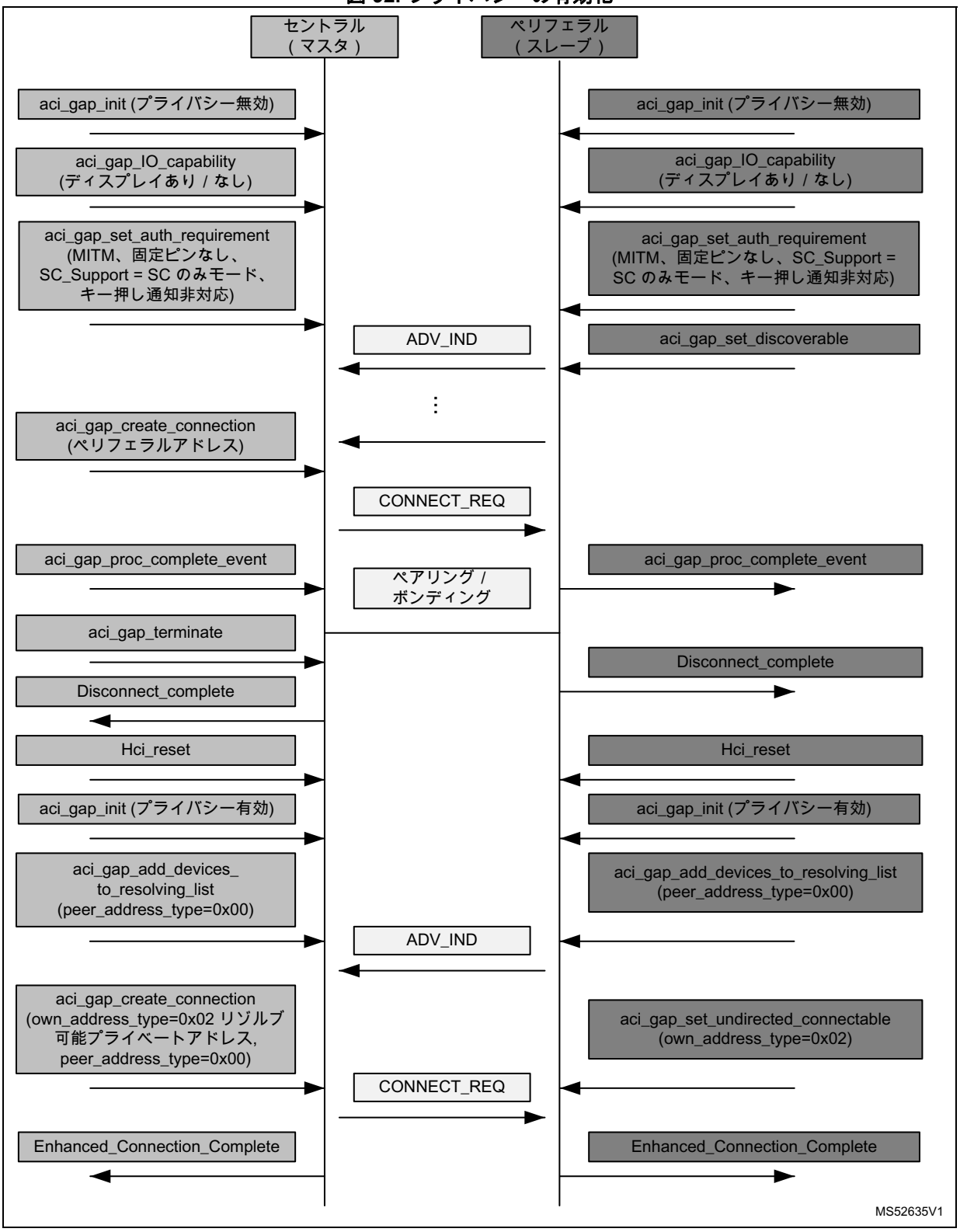


13.8.4 プライバシーの有効化

図 92 に、プライバシーが有効化されているデバイスに対する接続、ペアリング、ボンディング、切断、リセット、再初期化のシーケンスを示します。

参考資料

図 92. プライバシーの有効化



13.9 Bluetooth LE - リンク･レイヤ･データ･パケット

Bluetooth LE には、アドバタイジング･パケットとデータ･チャネル･パケットの両方に使用される1つのパケット･フォーマットがあります。

図 93. データパケットの内訳

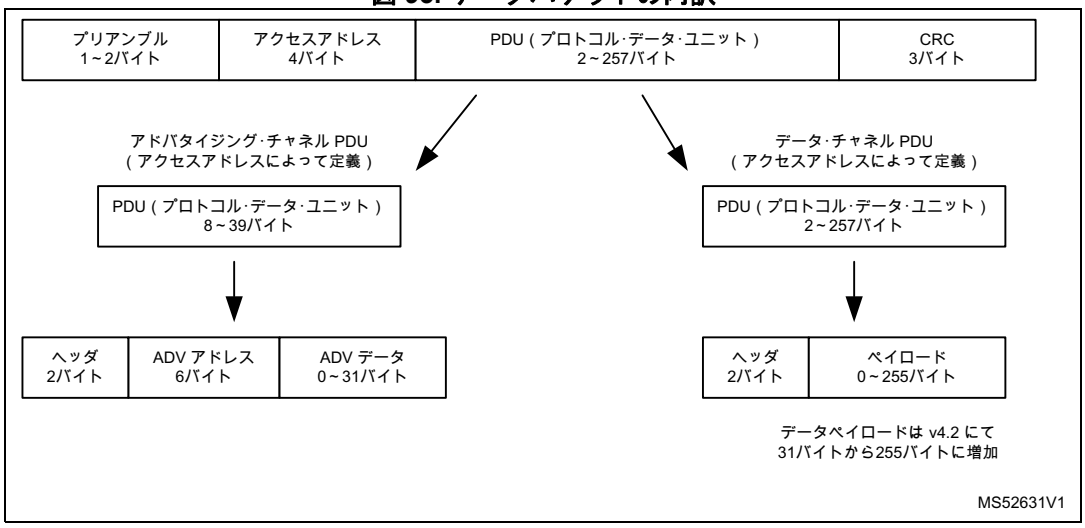
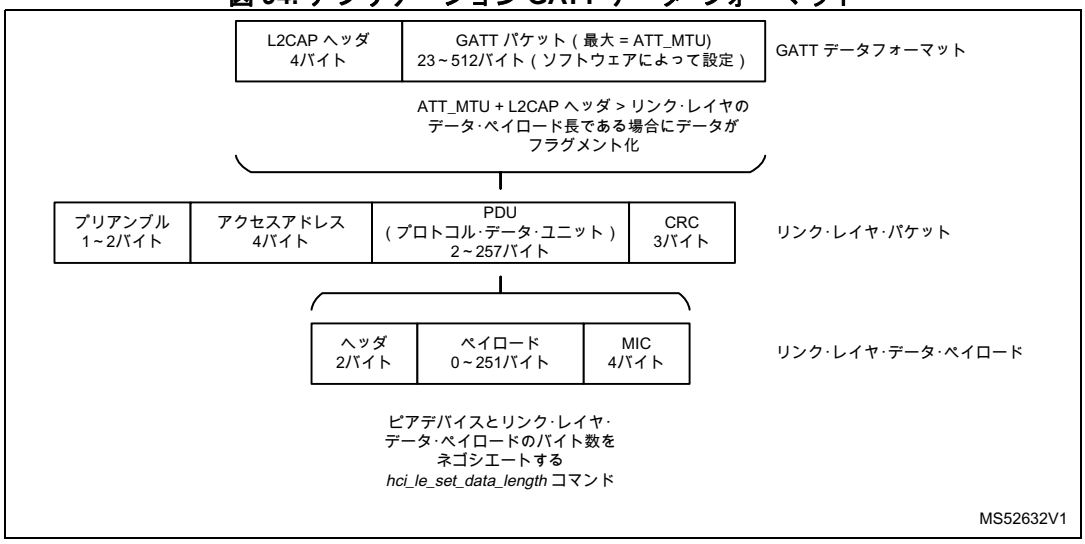


図 94. アプリケーション GATT データ･フォーマット



13.10 Thread の概要

13.10.1 概要

Thread スタックは高信頼性、低コスト、低消費電力なワイヤレス D2D 通信のためのオープン･スタンダードです。この規格は、IP ベースのネットワークが要求され、各種のアプリケーション･レイヤがスタックに使用される可能性のあるコネクテッド･ホーム･アプリケーション用として特別に設計されています。

仕様書一式（[9]）は http://threadgroup.org/ から入手可能です。

この規格は、IEEE 802.15.4 [IEEE802154] PHY（物理）レイヤと MAC（メディア･アクセス･コントロール）レイヤに基づいており、2.4GHz 帯で 250kbps の速度で動作します。

13.10.2 主な特徴

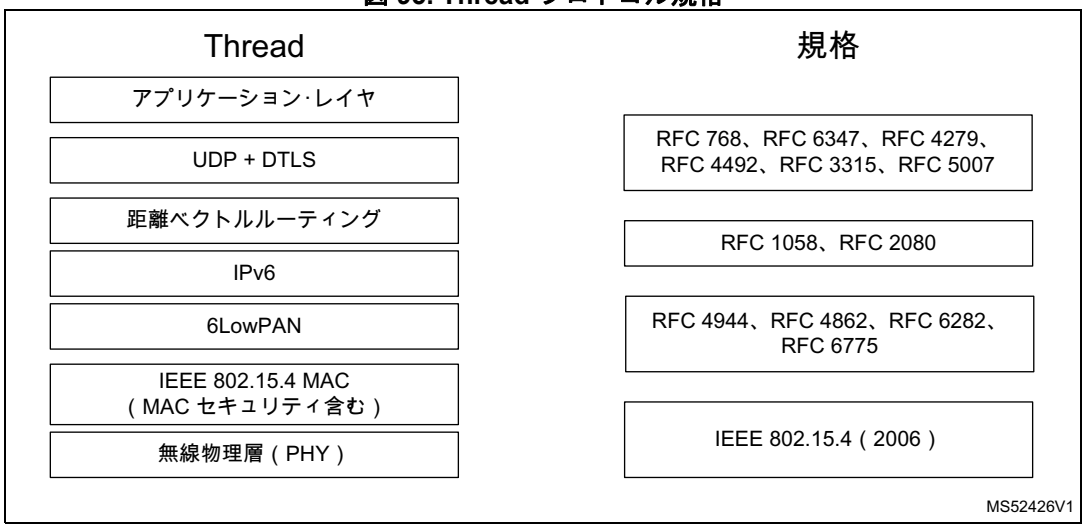
Thread は、環境制御、サーモスタット、アラーム、エネルギー管理、スマート･ロッカー、スマート照明器具のようなスマート･ホーム･アプリケーションをターゲットとしています。この規格の主な利点の 1つは IPv6 ベースであることです。そのため、あらゆる Thread ネットワークは、他の IPV6 アプリケーションに簡単に接続可能です。もう 1つの大きな利点は、真のメッシュ･ネットワークをベースにしていることです。ひとたび配備されると、このネットワークは非常に堅牢で高信頼性であると考えられます。たとえば、経路にエラーが生じた場合、システムは、転送先への新しい経路を見つけることにより、自分で自動設定が可能です。メッシュ･ネットワークを通じて、デバイスはより長距離におよぶ相互通信が可能です。

Thread には、アプリケーション･レイヤは何も定義されていません。ただし、大半の Thread アプリケーションは、CoAP を使用してデータ転送を行います。CoAP は広範囲に採用されており、Thread の中で、ネイティブで、たとえばアドレス解決管理にすでに使用されています。STM32WB デバイスでは、CoAP レイヤはユーザに公開されています。

13.10.3 レイヤ

Thread は、図 95 にある成熟した十分な実績のある規格に基づいています。

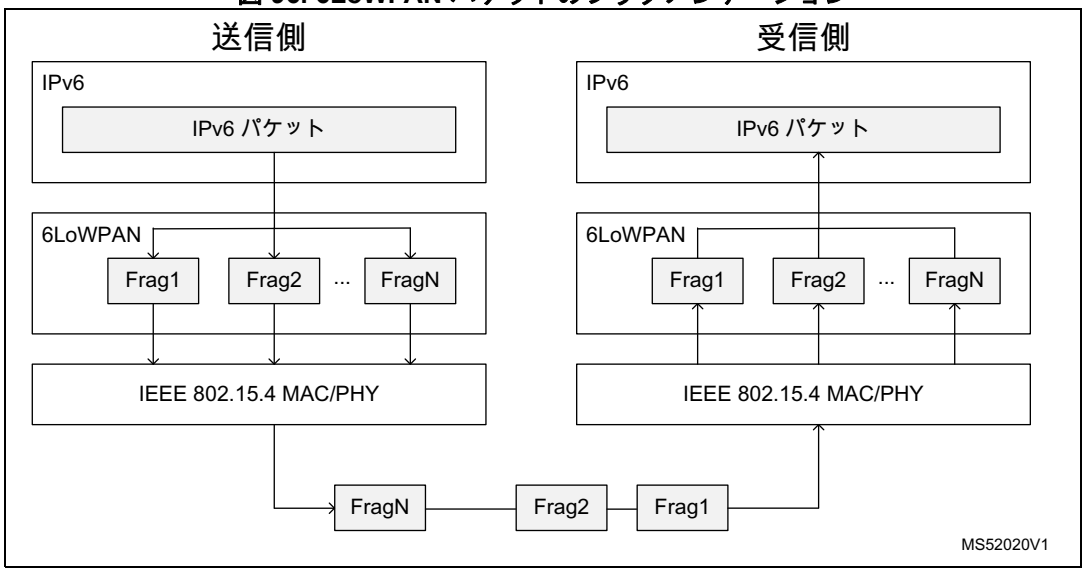
図 95. Thread プロトコル規格



下のレイヤから順に説明します。

- MAC レイヤは IEEE 802.15.4 仕様（2006 年版）のサブセットのみに基づいています。2.4GHz 帯で 250kbps のレートをサポートしています。チャネル 11からチャネル 26までの範囲の 16 チャネルが使用可能です。Thread ネットワークの内部では、1チャネルだけがリアルタイムで使用されます。MAC 無線レイヤは CSMA メカニズムを使用してフレームを送信します。伝送媒体がビジーである場合、ランダム値だけディレイさせられます。このメカニズムによって、2ノード間の伝送衝突の確率が低くなります。STM32WB では、ランダム値による伝送遅延は直接ハードウェアによって管理されます。
- 6LoWPAN レイヤ : 6LoWPAN は、「IPV6 over low power wireless personal area networks」を意味します。イーサネットでは、1280バイトの IPV6パケットは、１つの「モノリシック」フレームとして簡単に送信されます。MAC レイヤでは、最大パケットサイズが 127バイトに制限されているため、こうすることは可能ではありません。このため、Thread では 6LowPan レイヤを使用しています。このレイヤは次の 2つの技法を使用して実装されます。
  - フラグメンテーション（パケットを小さな TX 片に分割して RX にて再度組立て）
  - ヘッダ圧縮（場合により、48バイトのヘッダはわずか 6バイトのヘッダに圧縮可能）

図 96. 6LoWPAN パケットのフラグメンテーション



- IPv4を置き換える予定の IPv6（インターネット･プロトコル･バージョン 6）

  IPv4は32ビット･アドレッシングを使用しますが、IPV6は128ビット･アドレッシングを使用しますので、可能性は数十億倍になります。アドレッシング空間が大きくなるのに加えて、IPv6にはそれ以外にも技術的利点があります。特に、ルーティング手順が容易となります。

  Thread の中には、複数のアドレスが定義されています。

  - MeshLocal64 : アドレスは「トポロジ独立」であり、デバイスがルータやエンドデバイスとなったとしても、アドレスは一定であり絶対に変化しません。MeshLocal64 アドレスは、１台のデバイスから別のデバイスにPINGを送信する際に通常使用されるアドレスです。
  - MeshLocal16 : アプリケーションが下位レベルで mMeshLocal64 を使用する場合であっても、スタックはルーティングに mMeshLocal16 アドレスを使用します。Mesh Local には RLOC フィールド（ルーティング･ロケータ）が含まれています。アドレスはトポロジ依存（ネットワークとリンク品質に依存）です。子は、新しい親（新しいルータ）を選択して新しいアドレスを取得することができます。ユースケースによっては、子がルータになることも可能です。

  - MeshLinkLocal64：アドレスは 0xFE80 から始まり、ユニバーサル／ローカルビットを反転した MAC 拡張アドレスで終わります。ポイントリンクのダイレクト･ポイントや、MLE メッセージに使用されます。

- ルーティング：すべてのメッシュ･ネットワーク管理は MLE（メッシュ･リンク確立）メッセージに基づいています。これらのメッセージは、隣接デバイスの検出、システムの設定、ネットワーク全体のルーティング･コストの維持に使用されます。Thread は、非常に堅牢であり動的ルーティング適応の管理が可能であると言われています。ルータは、次のパラメータを含むアドバタイジングメッセージを定期的に送信します。
  - 送信側とその隣接の間のリンク品質
  - Thread ネットワーク･パーティション内のすべてのルータにアクセスするためのルート･コスト

  すべてのルータには、すべての隣接ルータとの UL と DL のリンク品質と、メッシュ･ネットワーク内部に存在するすべてのルータに対するルーティングコストのテーブルが含まれています。

  このテーブルには、ネットワーク全体を伝播する方法が定義されている「次ホップ」と、最後にアドバタイズメントを受信してからの経過時間を表す、いわゆる「経過時間」値も存在します。

  品質リンクは、0から3までで構成される値であり、3が最高品質（記録信号強度が 20dB 超）となります。取り得るリンク品質の値はわずか4種類であることから、隣接とリンク品質を通信するオーバーヘッドは最小限に抑えられます。リーダとして振る舞うルータは、ルータ ID の割当てと各ルータに関連付けられた拡張アドレスを追跡するための別のデータベースを保持しています。
- アプリケーション･レイヤ：Thread は CoAP をサポートしており、本設計では、このプロトコルはアプリケーション･レイヤとして振る舞います。CoAP は、非常に軽いバージョンの http プロトコルと見なすことができます。必要なリソースは http よりも遥かに少なく、オーバーヘッドも非常に小さくなっています。http 同様に、CoAP は REST モデルをベースとしています。サーバがある URL におけるリソースを利用可能とし、Get、Put、Post、Delete などのリクエストを使用してクライアントがこれらのリソースにアクセスします。URL（ユニフォーム･リソース･ロケータ）によって、リソースとそれへのアクセス方法が指定されます。メッセージには次の 4種類があります。
  - 確認不能メッセージ
  - 確認可能メッセージ
  - 確認応答メッセージ
  - リセット･メッセージ

13.10.4 メッシュ･トポロジ

Thread はメッシュ･ネットワークをサポートしています。図 97 に示すように、Thread ネットワーク内部のデバイスは次の２つの役割を持つことができます。

- ルータ
  - ネットワーク･デバイスにパケットを転送します。
  - ネットワークへの参加を試みるデバイスにセキュアなコミッショニング･サービスを提供します。
  - トランシーバを常時有効に保ちます。
- エンドデバイス
  - 主として １台のルータと通信を行います。
  - 他のネットワーク･デバイスにパケットを転送しません。
  - 消費電力低減のためにトランシーバを無効にできます。

すべてのルータの中で、常に 1 台が「リーダ」に格上げされます。Thread リーダとは、ある Thread ネットワークの中の一組のルータを管理するルータのことです。

すべてのエンドデバイスの中で、スリーピーエンドデバイス、REED、標準エンドデバイスが存在可能です。

- REED（ルータ適格エンドデバイス）とは、必要に応じてルータに格上げ可能なエンドデバイスのことです。
- スリーピーエンドデバイスは、通常は無効化されていて、不定期にウェイクアップしてその親からのメッセージをポーリングしたり、データを送信したりします。

メッシュ·ネットワークのサイズは設定可能です。最大で32台のアクティブなルータが存在します。各ルータは、異なる子デバイスに接続できます。各子 ID は 9ビットでコード化されており、理論上はルータ 1 台あたりの子の数は最大で 511台となります。STM32WB のメモリ制約により、ルータ 1 台あたりの子の数は10台に制限されています。

図 97. Thread ネットワークのトポロジ

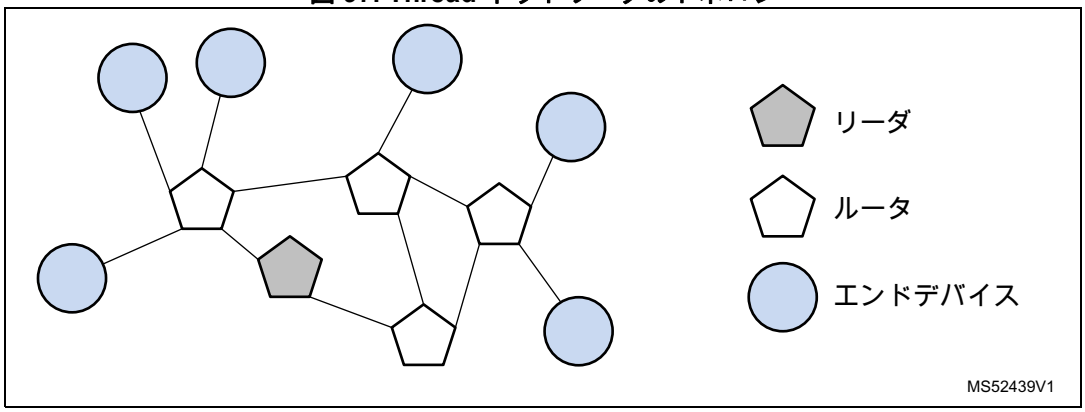
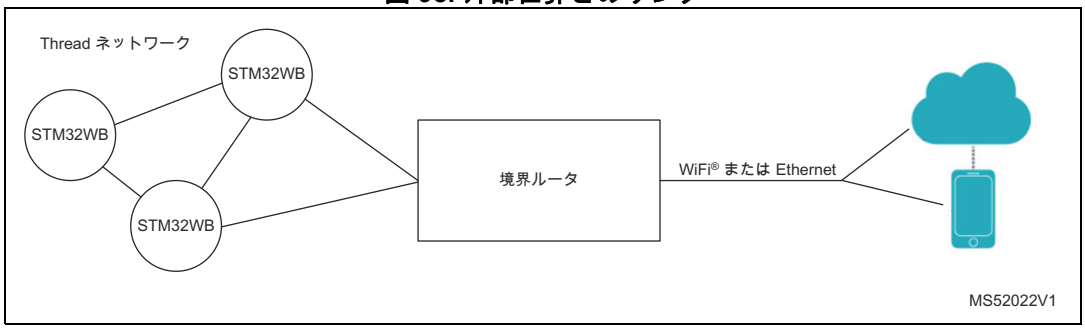


図 98. 外部世界とのリンク



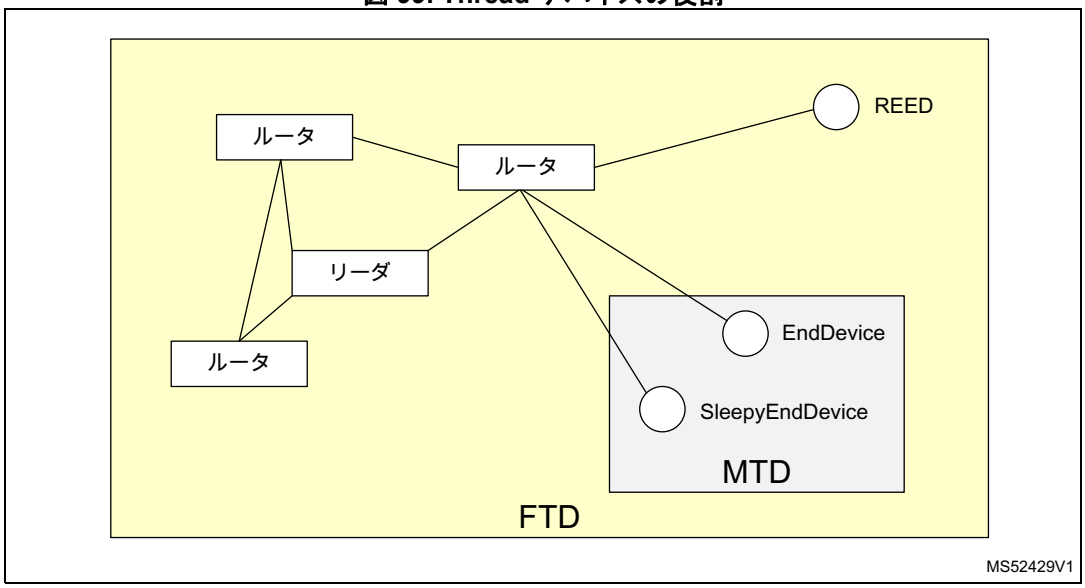
13.10.5 Thread の設定

OpenThread のコンパイル時に、ターゲットとしているユースケースによっていくつかのオプションを設定可能です。STM32WB では、次の2つのケースが考えられます。

1. フル Thread デバイス（FTD）: デバイスはシンプルなスリーピーエンドデバイスとして振る舞いますが、Thread ネットワーク内のルータやリーダとなることも可能です。
2. 最小 Thread デバイス（MTD）:
   デバイスは、エンドデバイスまたはスリーピーエンドデバイスのみとして振る舞います。

MTD 設定に必要なメモリ（RAM と Flash の両方）は、FTD 設定よりも少なくなります。その一方で、MTD はエンドデバイスまたはスリーピーエンドデバイスのみとして振る舞います。

図 99. Thread デバイスの役割



参考資料

14 まとめ

STM32WB シリーズ·マイクロコントローラをベースとする Bluetooth Low Energy（BLE）アプリケーションまたは802.15.4アプリケーションには、専用ソフトウェア·プロトコルとアーキテクチャの理解が必要になります。

このアプリケーション·ノートには、開発者が組み込みアプリケーションソフトウェアを作成するために必要な方法と、システム初期化の正しい手順に従うための主な要素が詳細に説明されています。

15 改版履歴

表 35. 文書改版履歴

日付	版	変更内容
2019 年 6 月 18 日	1	初版発行
2019 年 9 月 26 日	2	概要、セクション 4.2 : メモリマッピング、セクション 4.3 : 共有ペリフェラル、セクション 9.2 : 起動方法、セクション 9.6 : OpenThread API、セクション 11.1 : Thread_Cli_Cmd、セクション 11.4 : Thread_Coap_Multiboard、セクション 11.5 : Thread_Commissioning、セクション 12.3 : API、および セクション 12.4.3 : MAC アプリケーション･プロセッサ･ファームウェアを更新。 セクション 11.8 : Thread FUOTAおよびそのサブセクションを追加。 図 4 : メモリマッピングを更新。 表 2 : セマフォを更新。
2020 年 3 月 23 日	3	セクション 4.3 : 共有ペリフェラル および セクション 7.6.1 : Bluetooth デバイスアドレスの設定方法を更新。 セクション 4.7 : Flash メモリ管理、セクション 4.8 : CPU からのデバッグ情報、セクション 4.9 : FreeRTOS 低消費電力、およびそのサブセクションを追加。 表 2 : セマフォ、表 33 : システム･インタフェース･コマンドおよび表 34 : ユーザ･システム･イベントを更新。 図 10 : Flash メモリのデータを書き込み／消去するアルゴリズム、図 24 : 心拍数プロジェクト - ミドルウェアとユーザアプリケーションの相互作用および図 29 : P2P サーバ･ソフトウェア通信を更新。 ドキュメント全体で文章を軽微に編集。

表 36. 日本語版文書改版履歴

日付	版	変更内容
2021 年 3 月 31 日	1	日本語版初版発行

参考資料

AN5289

重要なお知らせ（よくお読み下さい）

STMicroelectronics NV およびその子会社（以下、ST）は、ST製品及び本書の内容をいつでも予告なく変更、修正、改善、改定及び改良する権利を留保します。購入される方は、発注前にST製品に関する最新の関連情報を必ず入手してください。ST 製品は、注文請書発行時点で有効なST の販売条件に従って販売されます。

ST製品の選択並びに使用については購入される方が全ての責任を負うものとします。購入される方の製品上の操作や設計に関してST は一切の責任を負いません。

明示又は黙示を問わず、STは本書においていかなる知的財産権の実施権も許諾致しません。

本書で説明されている情報とは異なる条件でST 製品が再販された場合、その製品についてSTが与えたいかなる保証も無効となります。

ST およびST ロゴはSTMicroelectronics の商標です。その他の製品またはサービスの名称は、それぞれの所有者に帰属します。

本書の情報は本書の以前のバージョンで提供された全ての情報に優先し、これに代わるものです。

この資料は、STMicroelectronics NV 並びにその子会社(以下ST)が英文で記述した資料（以下、「正規英語版資料」）を、皆様のご理解の一助として頂くためにSTマイクロエレクトロニクス㈱が英文から和文へ翻訳して作成したものです。この資料は現行の正規英語版資料の近時の更新に対応していない場合があります。この資料は、あくまでも正規英語版資料をご理解頂くための補助的参考資料のみにご利用下さい。この資料で説明される製品のご検討及びご採用にあたりましては、必ず最新の正規英語版資料を事前にご確認下さい。ST及びSTマイクロエレクトロニクス㈱は、現行の正規英語版資料の更新により製品に関する最新の情報を提供しているにも関わらず、当該英語版資料に対応した更新がなされていないこの資料の情報に基づいて発生した問題や障害などにつきましては如何なる責任も負いません。