RA4W1 グループ

Bluetooth Mesh 開発ガイド

要旨

Bluetooth Mesh スタックは、Bluetooth Mesh Networking 仕様に準拠した Mesh ネットワークを構成して、多対多デバイス間で無線通信を実行するためのソフトウェアライブラリです。本書は Bluetooth Mesh スタックのソフトウェア構成と各レイヤーの概要、Mesh アプリケーションの開発方法について示します。Bluetooth Mesh スタックの導入方法については「*RA4W1 グループ Bluetooth Mesh スタートアップガイド*」(R01AN5847)を参照してください。

対象デバイス

RA4W1 グループ

関連文書

下記の文書がルネサスのウェブサイトにて公開されています。

文書名	文書番号
RA4W1 グループ ユーザーズマニュアル ハードウェア編	R01UH0883
Renesas Flexible Software Package User's Manual	-
RA4W1 Group Bluetooth Low Energy Application Developer's Guide	R01AN5653
RA4W1 グループ Bluetooth Mesh スタートアップガイド	R01AN5847
RA4W1 グループ Bluetooth Mesh サンプルアプリケーション アプリケーションノート	R01AN5848
RA4W1 グループ Bluetooth Mesh 開発ガイド	R01AN5849 本書

目次

1. Bluetooth Mesh 概要 4
1.1 ノード 4
1.2 エレメント 4
1.3 アドレス 5
1.4 ステート 5
1.5 モデル 6
1.5.1 クライアントとサーバー 6
1.5.2 ファウンデーションモデル 6
1.5.3 コンフィグレーションモデル 7
1.5.4 ヘルスモデル 7
1.5.5 パブリケーションとサブスクリプション 7
1.6 メッセージ 8
1.7 Mesh ベアラー 9
1.8 プロビジョニング 10
1.9 コンフィグレーション 11
1.10 オプション機能 12
1.10.1 リレー 12
1.10.2 プロキシ 13
1.10.3 フレンドシップ 14

2. Mesh アプリケーション概要 15
2.1 ソフトウェア構成 15
2.2 ファイル構成 17
2.3 Mesh アプリケーション 18
2.3.1 Mesh コアモジュール 19
2.3.2 Mesh モデルモジュール 19
2.3.3 Mesh モデル構成 19
2.3.3.1 コンフィグレーションモデル 21
2.3.3.2 ヘルスモデル 23
2.3.3.3 Generic OnOff モデル 24
2.3.3.4 Vendor モデル 24
2.4 Bluetooth Mesh スタック 25
2.5 Bluetooth ベアラー 27
2.5.1 メッセージ送受信のためのベアラー関数 27
2.5.2 接続制御のためのベアラー関数 28
2.5.3 Mesh GATT サービス 29
2.5.4 ADV ベアラー動作 30
2.5.5 GATT ベアラー動作 31
2.6 MCU 周辺機能 32
2.7 Mesh サンプルプログラムの設定 35

2.8 Bluetooth ベアラーの設定 37

3. アプリケーション開発 38
3.1 メインルーチン 39
3.2 ノード構成の設定 42
3.3 プロビジョニング 43
3.3.1 プロビジョニングサーバー 43
3.3.2 プロビジョニングサーバーのシーケンス 46
3.4 プロキシ 50
3.4.1 プロキシサーバー 50
3.4.2 プロキシクライアント 52
3.4.3 プロキシのシーケンス 54
3.5 フレンドシップ 56
3.5.1 フレンドノード 56
3.5.2 ローパワーノード 57
3.5.3 ローパワーノードのシーケンス 60
3.5.4 フレンドノードのシーケンス 62
3.6 コンフィグレーション 64
3.6.1 コンフィグレーションサーバー 64
3.6.2 コンフィグレーションサーバーのシーケンス 65
3.7 ヘルスモデル 66
3.7.1 ヘルスサーバー 66
3.7.2 ヘルスサーバーのシーケンス 69
3.8 アプリケーションモデル 70
3.8.1 サーバーモデル 70
3.8.2 クライアントモデル 72
3.8.3 Generic OnOff モデルのシーケンス 74
3.8.4 Vendor Model のシーケンス 74

4. Appendix 75
4.1 Command Line Interface プログラム 75
4.2 プログラムサイズ 76

1. Bluetooth Mesh 概要

本章は Bluetooth Mesh Networking 仕様で定義された Bluetooth Mesh の基本的な概念について示します。本仕様の詳細は、Specifications List の Mesh Model 並びに Mesh Profile 仕様を参照してください。図 1-1 に Bluetooth Mesh ネットワークの基本構成を示します。

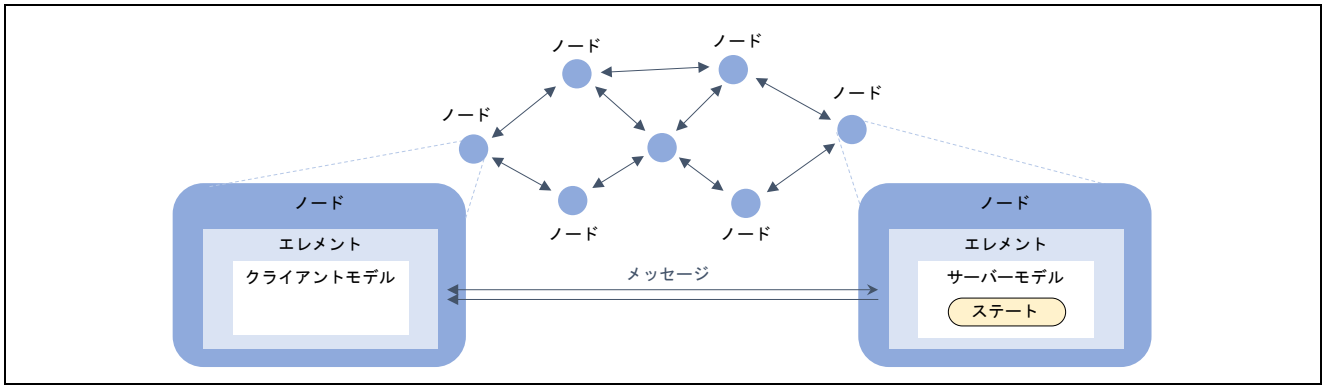

図 1-1　Bluetooth Mesh ネットワークの基本構成

1.1　ノード

ネットワークに参加しているデバイスはノード(Node)と呼ばれます。ネットワークは、アドレス空間と暗号鍵を共有するノードの集合です。ノード間の通信はネットワークキーによって暗号化されます。各ネットワークはネットワークキーから生成される Network ID で識別されます。デフォルトではプライマリサブネット(primary subnet)と呼ばれる 1 つのネットワークが構築されます。また通信対象を限定するために複数のサブネット(subnet)を構築することもできます。

1.2　エレメント

エレメント(Element)はノード内の論理的なオブジェクトです。ノードは少なくとも一つのエレメントを持ち、複数のエレメントを持つこともできます。最初のエレメントはプライマリエレメント(primary element)と呼ばれます。各エレメントはプロビジョニングによって割り当てられたユニキャストアドレスによって、ネットワーク上で一意に識別されます。

1.3 アドレス

ネットワークで使用されるアドレスは 16 ビット長です。アドレスタイプとして、未割り当てアドレス、ユニキャストアドレス、バーチャルアドレス、グループアドレスが定義されます。

表 1-1 アドレスタイプ

アドレスタイプ	値	値の範囲
未割り当てアドレス	0b0000000000000000	0x0000
ユニキャストアドレス	0b0xxxxxxxxxxxxxxx (0b0000000000000000を除く)	0x0001～0x7FFF
バーチャルアドレス	0b10xxxxxxxxxxxxxx	0x8000～0xBFFF
グループアドレス	0b11xxxxxxxxxxxxxx	0xC000～0xFFFF

- **未割り当てアドレス(Unassigned address)**
  未割り当てアドレスは、ユニキャストアドレスを割り当てられていないエレメントに設定されるアドレスです。メッセージの発信元アドレスや宛先アドレスとして使用することはできません。
- **ユニキャストアドレス(Unicast Address)**
  ユニキャストアドレスは、一つのエレメントを一意に識別するためのアドレスです。1 つのネットワークでは 32,767 個のユニキャストアドレスを使用できます。メッセージの発信元アドレスや宛先アドレスとして使用できます。
- **バーチャルアドレス(Virtual Address)**
  バーチャルアドレスは、ラベル UUID(Label UUID)から生成されるマルチキャストアドレスです。メッセージの宛先アドレスとして使用できます。ラベル UUID は複数のエレメントを分類するための 128 ビット長の値です。この値は任意に生成することができ、OOB(Out-Of-Band)によってデバイス間で共有されます。またラベル UUID およびバーチャルアドレスを一元的に管理する必要はありません。
- **グループアドレス(Group address)**
  グループアドレスは一元的に管理され、用途に応じて動的に割り当てられるマルチキャストアドレスです。メッセージの宛先アドレスとして使用できます。また、表 1-2 に示す通り、全ノード(all-nodes)への同報等の用途向けに固定グループアドレス(Fixed Group Addresses)が定義されています。

表 1-2 固定グループアドレス

固定グループアドレス	値
all-proxies	0xFFFC
all-friends	0xFFFD
all-relays	0xFFFE
all-nodes	0xFFFF

1.4 ステート

ステート(State)は、エレメントの状態を示す値です。2 つ以上の値で構成されるステートはコンポジットステーツ(Composite States)と呼ばれます。さらに、他のステートに連動して変化するステートは、バウンドステーツ(Bound States)と呼ばれます。ステートは瞬時に、または時間を掛けて変化します。イニシャルステート(Initial State)からターゲットステート(Target State)までの時間は、トランジションタイム(Transition Time)と呼ばれます。またカレントステート(Current State)からターゲットステートまでの時間は、リメイニングタイム(Remaining Time)と呼ばれます。

1.5 モデル

モデル(Model)は、各ノードがアプリケーションのシナリオに従って動作するための基本的な機能を標準化したものです。モデルはステート、ステートに作用するメッセージ、関連する動作を定義します。

1.5.1 クライアントとサーバー

モデルはクライアント－サーバー構成を採ります。サーバーモデルはステートを持ちますが、クライアントモデルはステートを持ちません。サーバーモデルはクライアントモデルからメッセージを受信することにより、エレメントのステートを制御します。クライアントモデルは *GET* メッセージでサーバーモデルのステートを取得、*SET* メッセージや *SET Unacknowledged* メッセージでサーバーモデルに新しいステートを設定することができます。サーバーモデルはステートの変化を通知する場合や、*GET* メッセージや *SET* メッセージを受信した時の応答として、*STATUS* メッセージを送信します。サーバーモデルは *SET Unacknowledged* メッセージを受信した場合、*STATUS* メッセージを送信しません。

図 1-2 にノード構成を示します。ノードは複数のエレメントを持つことができます。エレメントは複数のモデルを持つことができますが、同一エレメント内に同種のモデルを持つことはできません

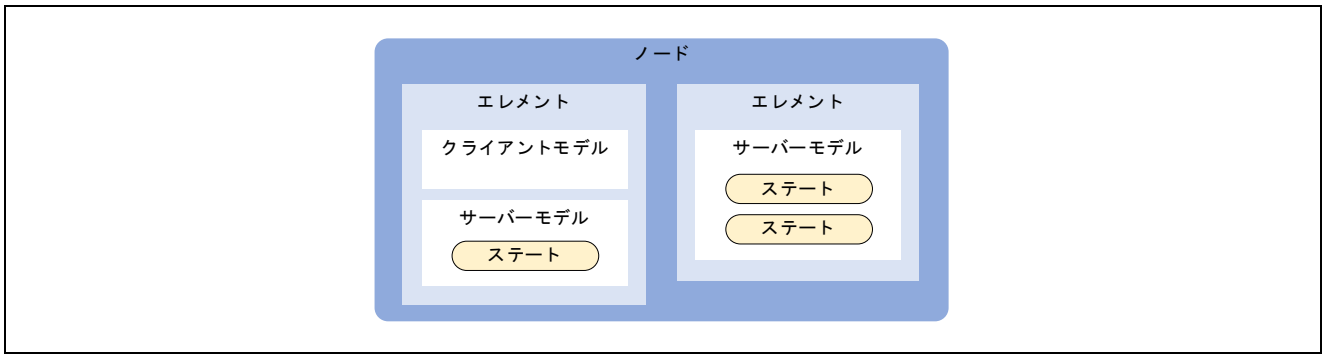

図 1-2 ノード構成

1.5.2 ファウンデーションモデル

ファウンデーションモデル(Foundation Models)は、各エレメントの動作を設定、管理するためのモデルです。各ノードのプライマリエレメントはコンフィグレーションサーバーモデル(Configuration Server Model)とヘルスサーバーモデル(Health Server Model)を持つ必要があります。

表 1-3 ファウンデーションモデル

モデル	SIG Model ID
Configuration Server	0x0000
Configuration Client	0x0001
Health Server	0x0002
Health Client	0x0003

1.5.3　コンフィグレーションモデル

コンフィグレーションモデル(Configuration Model)はノードの動作を設定するためのモデルです。ノードとエレメントの設定値はコンフィグレーションステートとして定義されます。コンフィグレーションサーバーモデル(Configuration Server Model)は、コンフィグレーションステート(Configuration State)を持つモデルです。コンフィグレーションクライアントモデル(Configuration Client Model)は、コンフィグレーションメッセージによって、コンフィグレーションサーバーの設定を管理するモデルです。コンフィグレーションメッセージはデバイスキー(Device Key)によって暗号化されます。デバイスキーは各ノードによって異なり、プロビジョニング時に生成されます。

1.5.4　ヘルスモデル

ヘルスモデル(Health Model)はノードの物理的な状態を監視するためのモデルです。ヘルスサーバーモデル(Health Server Model)は物理的な障害情報を表すフォールトステート(Fault State)を持つモデルです。ヘルスクライアントモデル(Health Client Model)はヘルスメッセージによって、ヘルスサーバーの障害情報を監視するモデルです。ヘルスメッセージはアプリケーションキー(Application Key)によって暗号化されます。

1.5.5　パブリケーションとサブスクリプション

モデルのメッセージ送信動作はパブリケーション(Publication)、メッセージ受信動作はサブスクリプション(Subscription)と呼ばれます。モデルは、マルチキャストアドレスを宛先アドレスとして設定することで、複数エレメントに対してメッセージをパブリッシュ(Publish)することができます。またモデルはマルチキャストアドレス宛のメッセージを選択的にサブスクライブ(Subscribe)することができます。図 1-3 にモデルによるメッセージのパブリッシュとサブスクライブの様子を示します。各モデルはモデルパブリケーションステートのパブリッシュアドレスに従ってメッセージを送信します。パブリッシュアドレスがマルチキャストアドレスの場合、各メッセージはサブスクリプションリストステートのサブスクリプションアドレスに従って複数のモデルからサブスクライブされます。

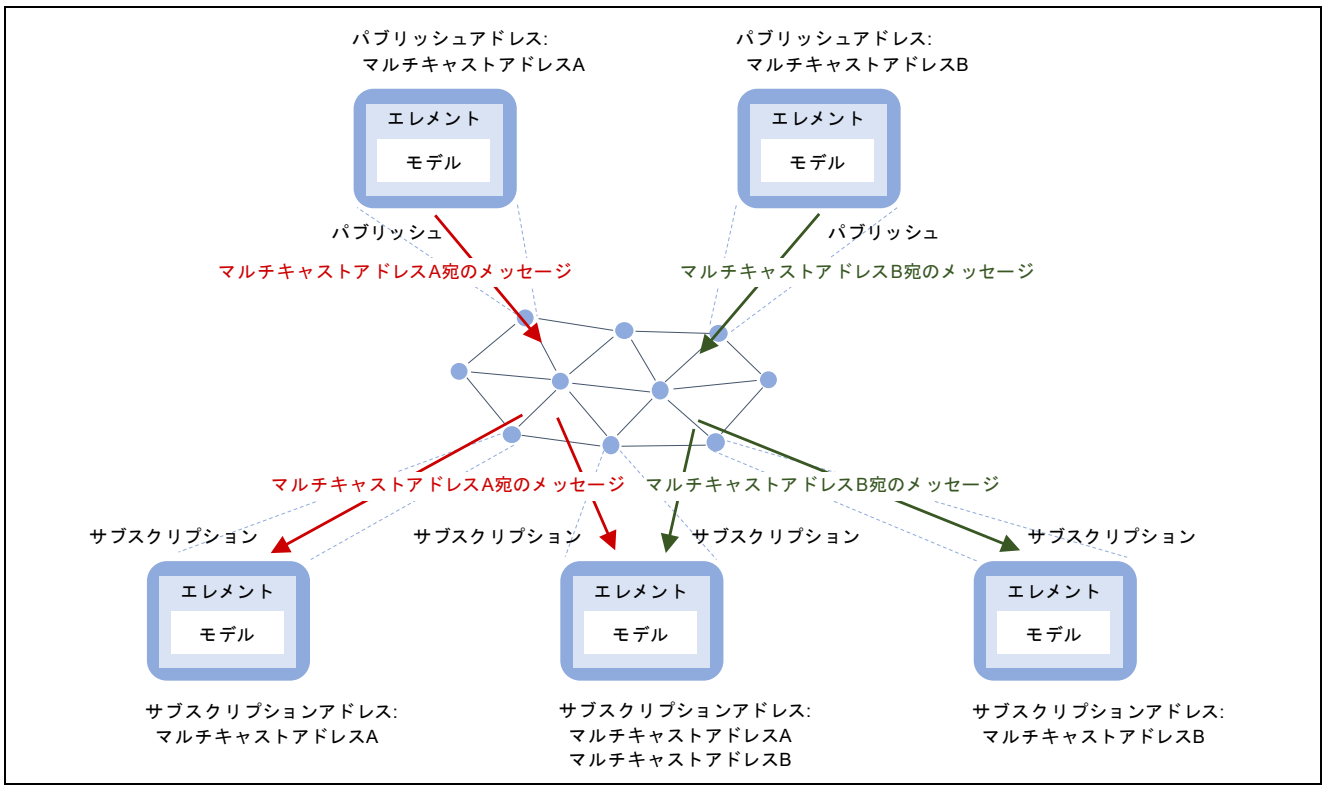

図 1-3　モデルによるメッセージのパブリッシュとサブスクライブ

1.6 メッセージ

ネットワークのノードによって送受信されるデータはメッセージ(Message)と呼ばれます。メッセージが複数にセグメント化されているか否かによって以下の様に分類されています。

- **Unsegmented メッセージ**

  Unsegmented メッセージは、セグメント化されていないデータを転送するためのメッセージです。11byte までの Access PDU を転送できます。

- **Segmented メッセージ**

  Segmented メッセージは、最大 32 セグメントまでセグメント化されたデータを転送するためのメッセージです。380byte までの Access PDU を転送できます。宛先ノードは全ての Segmented メッセージを受信後、データを再結合します。

メッセージの結合や分割は Bluetooth mesh の Access 層が行います。図 1-4 に Access 層における PDU のセグメント化と再結合を示します。各ノードは Network PDU を Mesh メッセージとして送受信します。

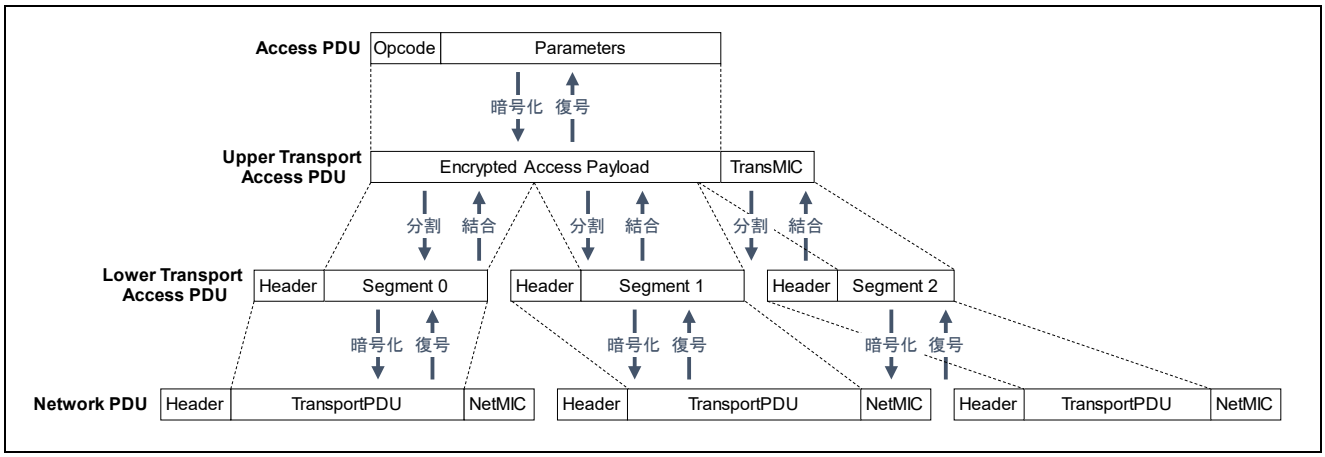

図 1-4 Access PDU のセグメント化と再結合

Network PDU のヘッダは発信元アドレス(SRC)や宛先アドレス(*DST*)、シーケンス番号(*SEQ*)といったフィールドを含みます。Network PDU はネットワークキー(Network Key)で暗号化されるため、同一のネットワークに参加しているノードだけがメッセージを復号化することができます。また、発信元アドレスや宛先アドレスは難読化(Obfuscation)されるため、ネットワークキーを持たないデバイスにそれらを特定されることはありません。

Lower Transport PDU のヘッダは Segmented か Unsegmented かを示す *SEG* やセグメント化されたデータを再結合するための SeqZero、SegO、SegN といったフィールドを含みます。

Access PDU はアプリケーションオペコード(Opcode)とアプリケーションパラメータ(Parameters)の 2 つのフィールドで構成されます。また Access PDU はアプリケーションキー(Application Key)またはデバイスキーで暗号化されます。これによってアプリケーションキーまたはデバイスキーを共有するノード間でのみ、データを共有することができます。アプリケーションキーはコンフィグレーションクライアントによって生成、配布されます。

1.7 Mesh ベアラー

Mesh ベアラー(Mesh Bearer)は、Mesh ネットワークにおいてメッセージを運搬する手段です。Bluetooth Low Energy 技術を利用した 2 つのベアラーが定義されます。

- **ADV ベアラー**

  ADV ベアラーは Non-Connectable and Non-Scannable Undirected Advertising でメッセージを送信します。本ベアラーが送信したメッセージは同時に複数のノードが受信できます。なお、プロビジョニングにおいてアドバタイズチャネルを用いてプロビジョニング PDU を運搬する場合、本ベアラーは PB-ADV と呼ばれます。

- **GATT ベアラー**

  GATT ベアラーは GATT サービスを通してメッセージを送信します。クライアント側のノードは Write Without Response によってメッセージを送信し、サーバー側のノードは Notification によってメッセージを送信します。GATT サービスを介して通信する前に、接続を確立する必要があります。本ベアラーが送信したメッセージは接続された対向ノードのみが受信できます。なお、プロビジョニングにおいてデータチャネルを用いてプロビジョニング PDU を運搬する場合、本ベアラーは PB-GATT と呼ばれます。

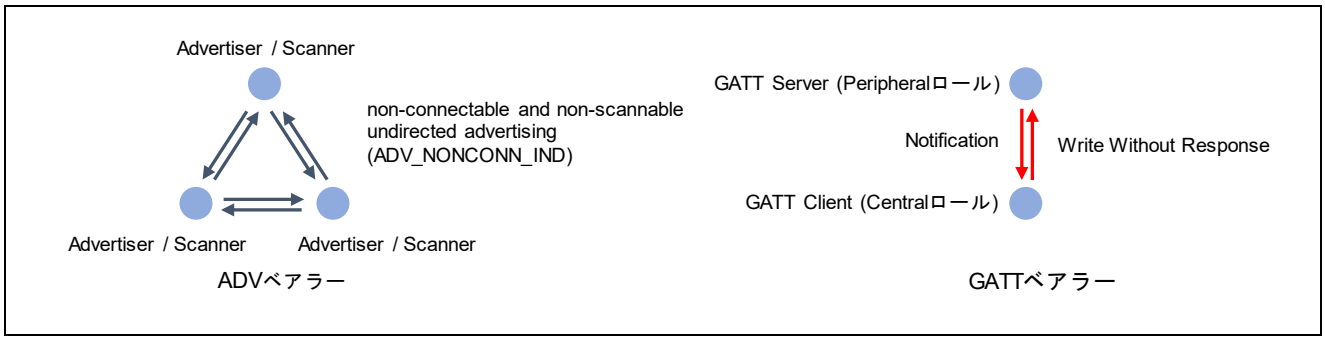

図 1-5 ADV ベアラーと GATT ベアラー

1.8 プロビジョニング

プロビジョニングは Mesh ネットワークに参加するための処理です。プロビジョニングではプロビジョニングデータが配布されます。プロビジョニングデータは下記の情報を含みます。

- ネットワークキーとネットワークキーインデックス
- キーリフレッシュフラグと IV アップデートフラグ
- カレント IV インデックス
- プライマリエレメントのユニキャストアドレス

Mesh ネットワークに参加していないデバイスはアンプロビジョンドデバイス(Unprovisioned Device)と呼ばれます。各アンプロビジョンドデバイスは 128 ビットデバイス UUID(Device UUID)によって識別されます。

デバイスを Mesh ネットワークに招待してプロビジョニングデータを配布するデバイスは、プロビジョニングクライアント(Provisioning Client)またはプロビジョナー(Provisioner)と呼ばれます。一般的に、プロビジョニングクライアントはスマートフォンまたはモバイル端末です。

プロビジョニングデータを受け取って Mesh ネットワークに参加するデバイスは、プロビジョニングサーバー(Provisioning Server)またはプロビジョニー(Provisionee)と呼ばれます。Mesh ネットワークへの参加が完了したデバイスはノードと呼ばれます。

1.9 コンフィグレーション

ノードがモデルを使用して他ノードと通信するためには、コンフィグレーション(Configuration)が必要です。コンフィグレーションによって、アプリケーションキーやパブリッシュアドレス、サブスクリプションアドレスといったモデル動作に必要な情報がノードに設定されます。図 1-6 にノードの典型的なライフサイクルを示します。

新規導入されたデバイスはプロビジョナーによってプロビジョニングされることで、Mesh ネットワークに参加します。さらにコンフィグレーションクライアントによってコンフィグレーションされることで、他ノードとの Mesh モデルによる通信が可能となります。一般的に、コンフィグレーションクライアントはスマートフォンまたはモバイル端末です。

ノードを Mesh ネットワークから除外する場合、コンフィグレーションクライアントは除去したいノードに対して Config Node Reset メッセージを送信する必要があります。この時、コンフィグレーションクライアントはネットワークで使用される暗号鍵を更新するため、除外されたノードは他ノードとの通信が不可能となります。

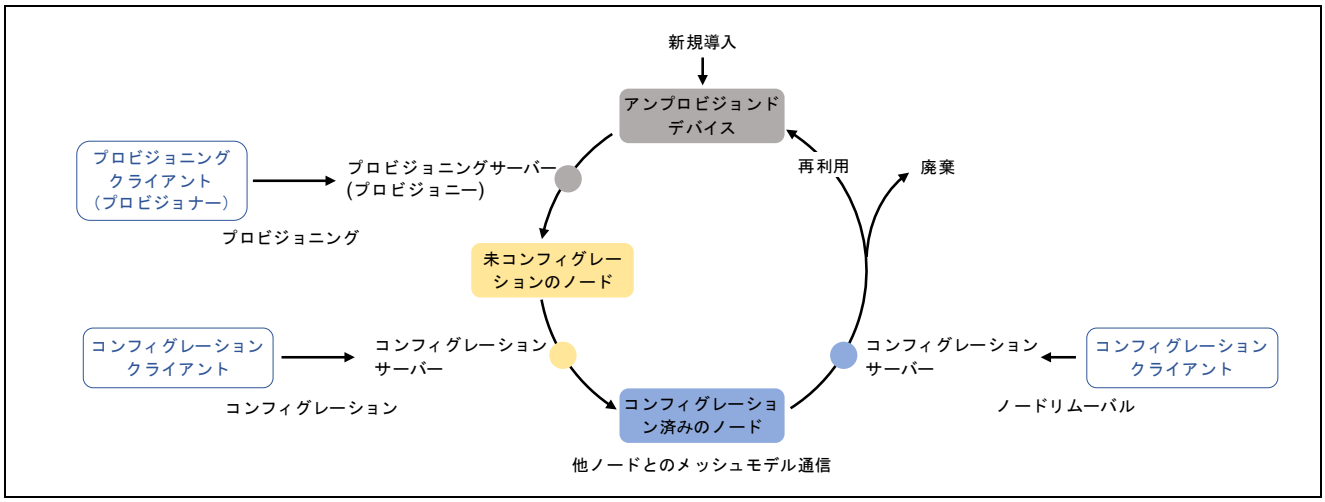

図 1-6 ノードのライフサイクル

1.10 オプション機能

下記の機能がオプション機能として定義されます。

- リレー機能 (1.10.1 項を参照)
- プロキシ機能 (1.10.2 項を参照)
- フレンド機能 (1.10.3 項を参照)
- ローパワー機能 (1.10.3 項を参照)

ノードが各オプション機能を有効化することで、多様な Mesh ネットワークを形成できます。次節から各オプション機能について記載します。

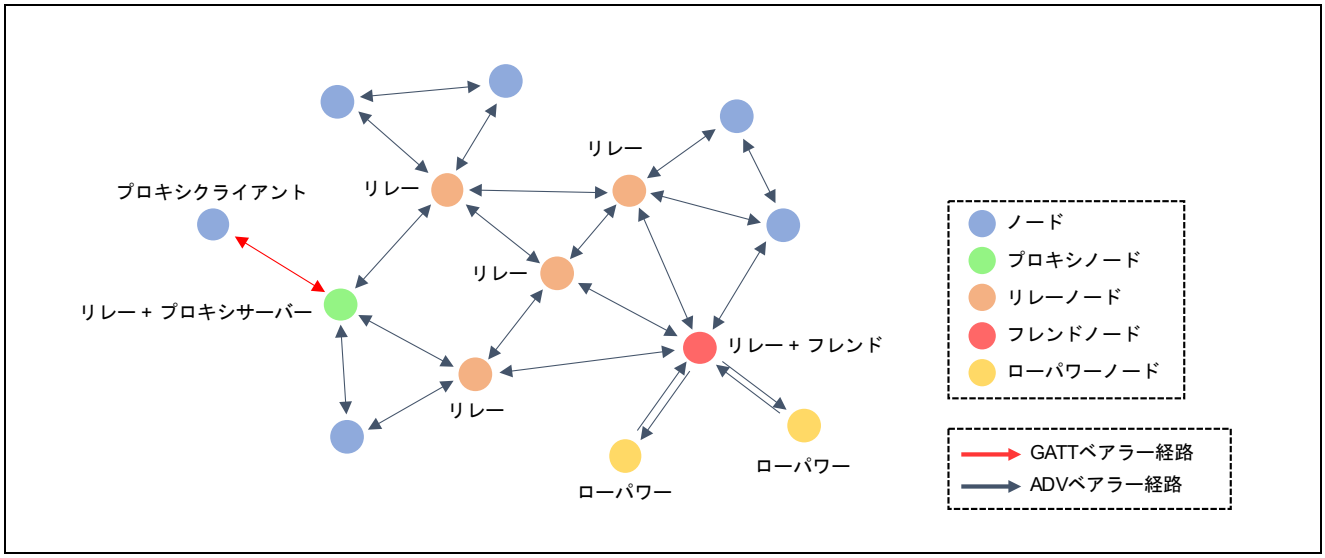

図 1-7　Mesh ネットワーク

1.10.1 リレー

リレー機能(Relay feature)は ADV ベアラーをサポートするノードが、受信したメッセージを中継する機能です。この機能により、宛先ノードが発信ノードの無線到達範囲外であっても、メッセージを他のノードが次々と中継してネットワーク中に拡散することで、最終的にメッセージは宛先ノードに到達することができます。メッセージを中継するノードは、リレーノード(Relay node)と呼ばれます。

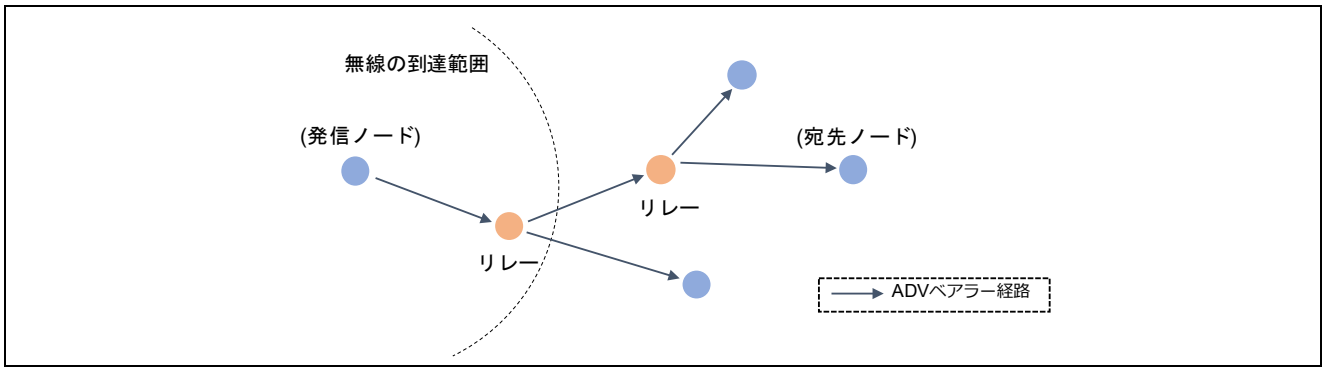

図 1-8　メッセージリレー

1.10.2 プロキシ

プロキシ機能(Proxy)は GATT ベアラーと ADV ベアラーの両方をサポートするノードが、GATT ベアラーと ADV ベアラー間でメッセージを転送する機能です。

GATT ベアラーのみに対応するノードは、接続した対向ノードとしか通信できません。その様なノードが Mesh ネットワークとの間でメッセージ授受を行う場合、プロキシ機能に対応するノードと接続を確立します。本ノードが送信したメッセージは、プロキシノードが ADV ベアラーに転送し、最終的に宛先ノードに到達することができます。さらに他ノードが送信したメッセージは、プロキシノードが GATT ベアラーに転送し、本ノードに到達することができます。GATT ベアラーと ADV ベアラー間でメッセージを転送するノードはプロキシサーバー(Proxy Server)と呼ばれます。またプロキシサーバーと接続して GATT ベアラーでメッセージを送受信するノードはプロキシクライアント(Proxy Client)と呼ばれます。

プロキシサーバーはプロキシクライアントのサブスクリプションアドレスを管理するためのリストを持ち、このリストはプロキシフィルタリストと呼ばれます。プロキシフィルタタイプとして、ホワイトリストフィルタまたはブラックリストフィルタを選択できます。プロキシフィルタタイプがホワイトリストフィルタの場合、プロキシサーバーはリストに登録されたアドレス宛のメッセージのみをプロキシクライアントに転送します。プロキシフィルタタイプがブラックリストフィルタの場合、プロキシサーバーはリストに登録されたアドレス宛のメッセージはプロキシクライアントに転送しません。

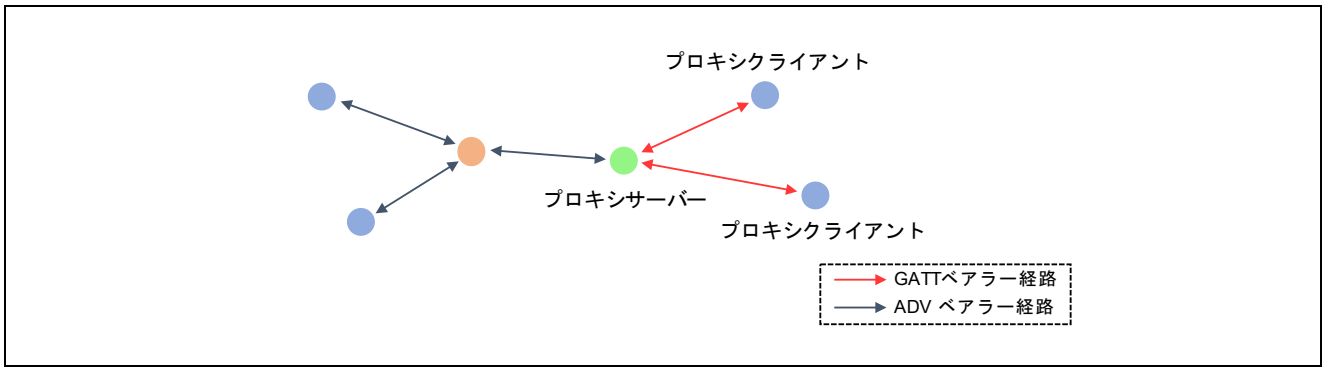

図 1-9　プロキシ

1.10.3 フレンドシップ

フレンド機能はローパワーノード宛のメッセージを保持し、ローパワーノードが要求するタイミングで転送する機能です。通常、ADV ベアラーに対応するノードは、メッセージを含む Advertising パケットを受信するため、常に Scan を実行します。ローパワー機能(Low Power)は、この Scan の実行時間を削減するための機能です。ローパワー機能に対応したノードは、Scan を休止することで消費電力を低減することができます。

ローパワー機能を利用するためには、フレンド機能(Friend feature)に対応したノードとフレンドシップ(Friendship)を確立する必要があります。フレンドシップを確立するためには、ローパワーノードはフレンドノードに対して、フレンドとなることを要求します。フレンドノードがこれを承認することで、フレンドシップが確立します。フレンドシップの確立後、ローパワーノードは Scan を停止することができる一方、フレンドノードはローパワーノードが必要とするメッセージを保持しなければなりません。

フレンドノードは、ローパワーノードのサブスクリプションアドレスを管理するためのリストを持ち、このリストはフレンドサブスクリプションリストと呼ばれます。フレンドノードはフレンドシップの確立後、リストに登録されたマルチキャストアドレス宛のメッセージを保持します。

ローパワーノードは、フレンドノードにメッセージが保持されているかを間欠的に問い合わせ、問い合わせの期間だけ Scan を再開します。フレンドノードは保持しているローパワーノード宛のメッセージをこのタイミングで転送します。

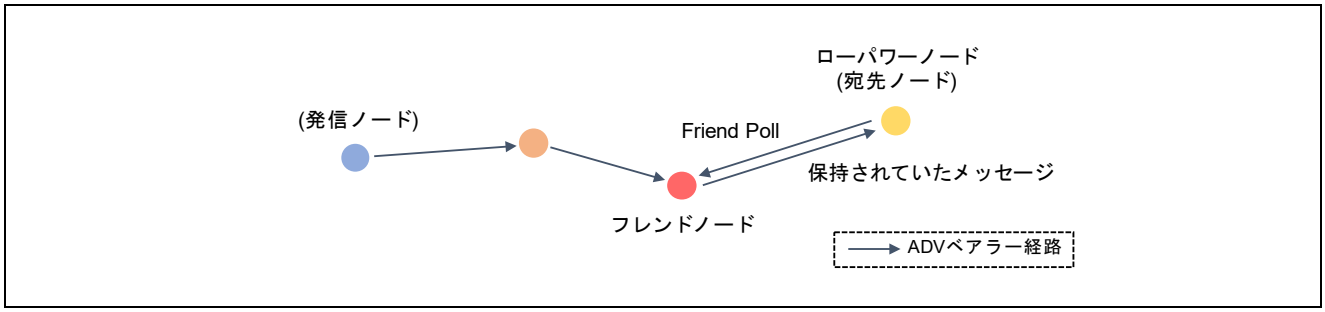

図 1-10　フレンドシップ

2. Mesh アプリケーション概要

本章は Mesh アプリケーションの概要を説明します。

2.1 ソフトウェア構成

図 2-1 に Mesh スタックを使用するソフトウェアの構成を示します。

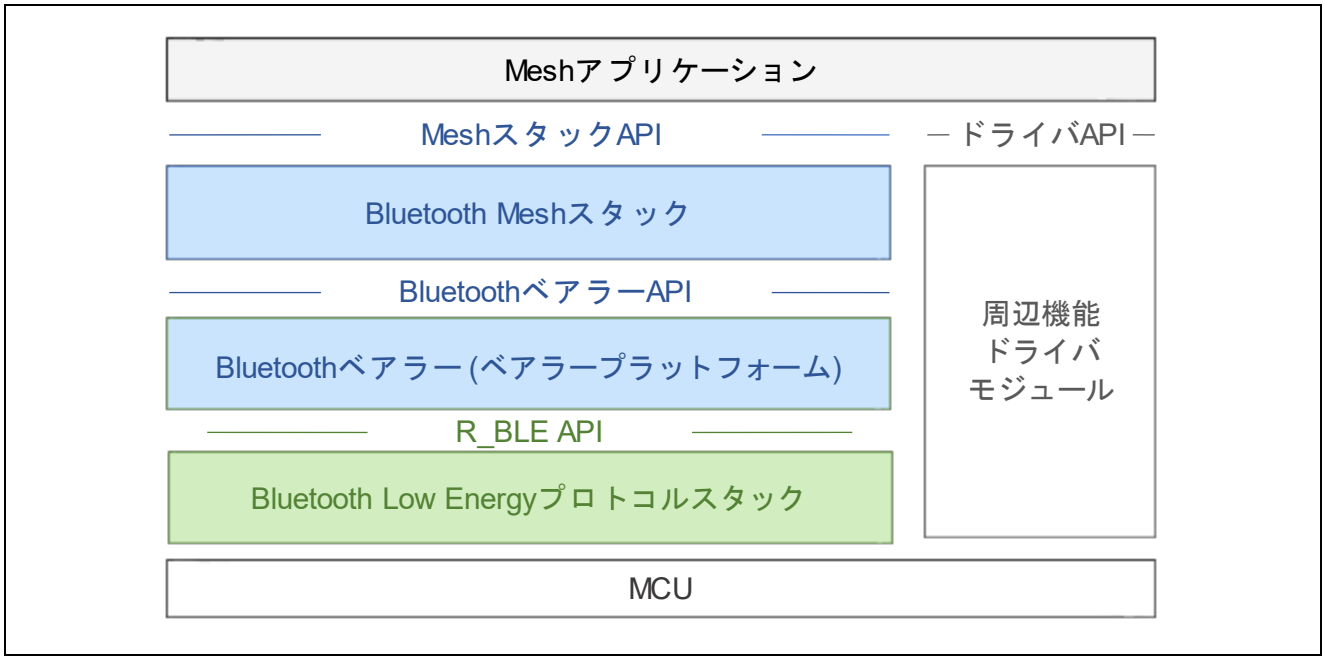

図 2-1 ソフトウェア構成

Mesh スタックを使用するソフトウェアは下記で構成されます。

- **Mesh アプリケーション**

  Mesh アプリケーションは、Bluetooth Mesh の無線通信機能を利用するアプリケーションプログラムです。ユーザは Mesh スタック API (*RM_BLE_MESH_XXXX, RM_MESH_XXXX*)や Bluetooth ベアラー API (*RM_BLE_MESH_BEARER_XXXX*)の仕様を理解し、Mesh アプリケーションを開発する必要があります。なお Mesh アプリケーションのサンプルプログラムは「RA4W1 グループ Bluetooth Mesh サンプルアプリケーション」(R01AN5848)に含まれます。

- **Bluetooth Mesh スタック**

  Bluetooth Mesh スタック(以後、"Mesh スタック"と表記)は、Bluetooth Mesh Networking 仕様に準拠した多対多の無線通信機能をアプリケーションに提供するソフトウェアです。Mesh スタックは Mesh ネットワーク通信を利用するための Mesh スタック API を有します。なお Mesh スタックは Renesas Flexible Software Package (FSP)として提供される Mesh モジュールに含まれます。

- **Bluetooth ベアラー(ベアラープラットフォーム)**

  Bluetooth ベアラーは、Bluetooth Low Energy プロトコルスタックのラッパー関数を Mesh スタックとアプリケーションに提供する抽象化レイヤーです。Bluetooth ベアラーもまた Renesas Flexible Software Package (FSP)として提供されるベアラープラットフォームモジュールに含まれます。

- **Bluetooth Low Energy プロトコルスタック**

  Bluetooth Low Energy プロトコルスタック(以後、"Bluetooth LE スタック"と表記)は、Bluetooth Low Energy 仕様に準拠した無線通信機能を上位レイヤーに提供するソフトウェアです。Bluetooth LE スタックは Bluetooth Low Energy 通信を利用するための R_BLE_API を有します。なお Bluetooth LE スタックは Renesas Flexible Software Package (FSP)として提供される BLE モジュールに含まれます。

- **周辺機能ドライバモジュール**

  アプリケーション、Mesh スタック、Bluetooth LE スタックはマイコンの周辺機能を利用します。RA マイコンのソフトウェア開発では Renesas Flexible Software Package (FSP)として提供される周辺機能ドライバを利用できます。

2.2 ファイル構成

「RA4W1 グループ Bluetooth Mesh サンプルアプリケーション」(R01AN5848)は下記のデモプロジェクトを含みます。

- ekra4w1_mesh_client_baremetal: EK-RA4W1 向け Client モデルプロジェクト(Baremetal)
- ekra4w1_mesh_server_baremetal: EK-RA4W1 向け Server モデルプロジェクト(Baremetal)
- ekra4w1_mesh_cli_client_baremetal: EK-RA4W1 向け Command Line Interface Client モデルプロジェクト
- ekra4w1_mesh_cli_server_baremetal: EK-RA4W1 向け Command Line Interface Server モデルプロジェクト
- ekra4w1_mesh_client_freertos: EK-RA4W1 向け Client モデルプロジェクト(FreeRTOS)
- ekra4w1_mesh_server_freertos: EK-RA4W1 向け Server モデルプロジェクト(FreeRTOS)

上記デモプロジェクトは、サンプルプログラムのビルドに必要な Mesh スタック、Bluetooth ベアラー、Bluetooth LE スタック、その他のモジュールを含みます。デモプロジェクトの構成を以下に示します。本書では**太字**で示すソフトウェアについて解説します。その他のモジュールの詳細は「Renesas Flexible Software Package User's Manual」を参照してください。

```
{project}\                                  :
    +---ra\fsp\lib\                         :  Mesh スタックライブラリ
    |                                       :  Bluetooth LE スタックライブラリ
    +---ra_cfg\                             :  モジュール設定
    |                                       :
    +---ra_gen\                             :  MCU 端子設定、ベクタテーブル
    |                                       :
    +---src\                                :
        |   app_main.c                      :  Mesh サンプルプログラム
        |   mesh_appl.h                     :  Mesh サンプルヘッダ
        |   mesh_core.c                     :  Mesh コアモジュール
        |   mesh_model.c                    :  Mesh モデルモジュール
        |                                   :
        +---app_lib\                        :  アプリケーションライブラリ
        +---mesh_cli\                       :  アプリケーションライブラリ
        |                                   :
        +---vendor_model\                   :
        |       vendor_api.h                :  Vendor モデルヘッダ
        |       vendor_client.c                Vendor Client モジュール
        |       vendor_server.c                Vendor Server モジュール
```

サンプルプログラムのビルド環境構築については、「RA4W1 グループ Bluetooth Mesh サンプルアプリケーション アプリケーションノート」(R01AN5848)の 2 章を参照してください。

2.3 Mesh アプリケーション

Bluetooth Mesh の無線通信機能を利用する Mesh アプリケーションは、ユーザが開発する必要があります。「RA4W1 グループ Bluetooth Mesh サンプルアプリケーション」(R01AN5848)は、Mesh アプリケーション開発のリファレンスとなるサンプルプログラムのソースコードを含みます。Mesh アプリケーションのサンプルプログラム(以後、"Mesh サンプルプログラム"と表記)は、Mesh スタック API を利用して、プロビジョニングと Mesh ノードとしての基本的な動作を実行します。Mesh サンプルプログラムは以下の機能を利用可能です。

- Unprovisioned Device 動作: PB-ADV ベアラーと PB-GATT ベアラーの両方に対応します。
- Configuration Server 動作: コンフィグレーション情報をデータフラッシュメモリに保存します。
- Generic OnOff Client 動作: RA4W1 向け開発ボードのスイッチ押下で Generic OnOff Set メッセージを送信します。
- Generic OnOff Server 動作: Generic OnOff Set メッセージの受信で RA4W1 向け開発ボードの LED を制御します。
- Vendor Client 動作: UART 経由で入力された文字列を Vendor Set メッセージで送信します。
- Vendor Server 動作: Vendor Set メッセージで受信した文字列を UART 経由で出力します。
- Low Power 動作: Friend ノードと Friendship を確立後、Friend Subscription List に Subscription アドレスを登録します。
- Proxy Server 動作: Proxy Client と接続を確立後、GATT ベアラー経由でメッセージを転送します。
- IV Update 開始機能: 送受信メッセージのシーケンス番号を監視し、閾値を超えた場合に IV Update を開始します。

本サンプルプログラムは次の 2 つのモジュールを含みます。これらのモジュールは Mesh サンプルプログラムの./src フォルダ以下に配置しています。

- **Mesh コアモジュール**

  本モジュールはプロビジョニングサーバーとしてプロビジョニングを実行し、プロビジョニングの完了後はプロキシサーバーとして GATT ベアラーを有効化します。また Low Power ノードとして Friendship を制御します。詳細は 2.3.1 項を参照してください。

- **Mesh モデルモジュール**

  本モジュールはコンフィグレーションサーバーモデルとヘルスサーバーモデルに加え、Generic OnOff モデルと独自 Vendor モデルに関する動作を実行します。詳細は 2.3.2 項を参照してください。

2.3.1 Mesh コアモジュール

Mesh サンプルプログラムに含まれる Mesh コアモジュールは次の処理を実行します。本モジュールは mesh_core.c に実装されています。

- プロビジョニング処理 (3.3 節を参照)
- プロキシ機能 (3.4 節を参照)
- ローパワー機能 (3.5 節を参照)
- IV Update 機能

2.3.2 Mesh モデルモジュール

Mesh サンプルプログラムに含まれる Mesh モデルモジュールは次の処理を実行します。本モジュールは mesh_model.c に実装されています。

- Mesh モデルの構成 (2.3.3 項を参照)
- コンフィグレーションモデル (2.3.3.1 項を参照)
- Generic OnOff モデル (2.3.3.3 項を参照)
- Vendor モデル (2.3.3.4 項を参照)

2.3.3 Mesh モデル構成

本サンプルプログラムでは次のモデルを使用します。

- Configuration Server モデル
- Health Server モデル
- Generic OnOff Server モデル (ONOFF_SERVER_MODEL マクロ*が定義されている場合に有効)
- Generic OnOff Client モデル (ONOFF_CLIENT_MODEL マクロ*が定義されている場合に有効)
- Vendor Server モデル (VENDOR_SERVER_MODEL マクロ*が定義されている場合に有効)
- Vendor Client モデル (VENDOR_CLIENT_MODEL マクロ*が定義されている場合に有効)

* これらのマクロはビルドオプションにて指定します。

図 2-2 に Mesh サンプルプログラムの各プロジェクトのモデル構成を示します。プライマリエレメントには Configuration Server モデル、Health Server モデルに加え Generic OnOff Server モデル、Generic OnOff Client モデル、Vendor Server モデル、Vendor Client モデルが配置されます。各モデルの説明を次節以降に記します。

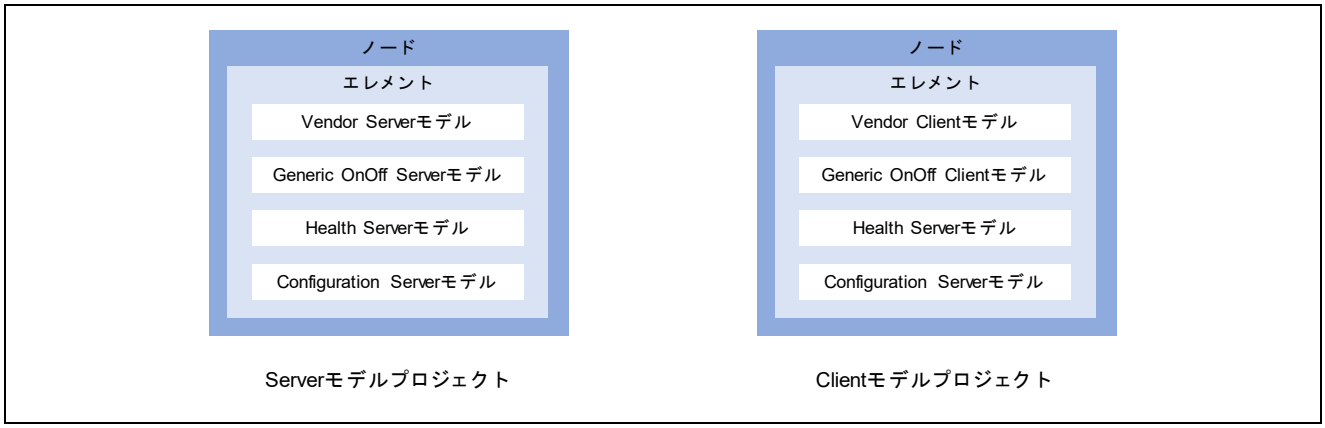

図 2-2 Mesh サンプルプログラムのモデル構成

2.3.3.1 コンフィグレーションモデル

コンフィグレーションモデルはノードの振る舞いを設定するための Mesh モデルです。コンフィグレーションサーバーはノードやエレメント、モデルの動作設定を保持する複数のコンフィグレーションステートを持ちます。これらのステートはコンフィグレーションクライアントからのメッセージによって操作されます。

表 2-1 コンフィグレーションモデルのステート

モデル名	SIG モデル ID	ステート
Configuration Server	0x0000	Secure Network Beacon Composition Data Default TTL GATT Proxy Friend Relay Model Publication Subscription List NetKey List AppKey List Model to AppKey List Node Identity Key Refresh Phase Heartbeat Publish Heartbeat Subscription Network Transmit Relay Retransmit PollTimeout List
Configuration Client	0x0001	-

表 2-2 コンフィグレーションメッセージ

ステート	メッセージ名	オペコード	方向
Secure Network Beacon	Config Beacon Get	0x8009	Client→Server
	Config Beacon Set	0x800A	Client→Server
	Config Beacon Status	0x800B	Server→Client
Composition Data	Config Composition Data Get	0x8008	Client→Server
	Config Composition Data Status	0x02	Server→Client
Default TTL	Config Default TTL Get	0x800C	Client→Server
	Config Default TTL Set	0x800D	Client→Server
	Config Default TTL Status	0x800E	Server→Client
GATT Proxy	Config GATT Proxy Get	0x8012	Client→Server
	Config GATT Proxy Set	0x8013	Client→Server
	Config GATT Proxy Status	0x8014	Server→Client
Friend	Config Friend Get	0x800F	Client→Server
	Config Friend Set	0x8010	Client→Server
	Config Friend Status	0x8011	Server→Client
Relay Relay Retransmit	Config Relay Get	0x8026	Client→Server
	Config Relay Set	0x8027	Client→Server
	Config Relay Status	0x8028	Server→Client
Model Publication	Config Model Publication Get	0x8018	Client→Server
	Config Model Publication Set	0x03	Client→Server
	Config Model Publication Virtual Address Set	0x801A	Client→Server
	Config Model Publication Status	0x8019	Server→Client
Subscription List	Config Model Subscription Add	0x801B	Client→Server
	Config Model Subscription Virtual Address Add	0x8020	Client→Server
	Config Model Subscription Delete	0x801C	Client→Server
	Config Model Subscription Virtual Address Delete	0x8021	Client→Server
	Config Model Subscription Virtual Address Overwrite	0x8022	Client→Server
	Config Model Subscription Overwrite	0x801E	Client→Server
	Config Model Subscription Delete All	0x801D	Client→Server

ステート	メッセージ名	オペコード	方向
	Config Model Subscription Status	0x801F	Server→Client
	Config SIG Model Subscription Get	0x8029	Client→Server
	Config SIG Model Subscription List	0x802A	Server→Client
	Config Vendor Model Subscription Get	0x802B	Client→Server
	Config Vendor Model Subscription List	0x802C	Server→Client
NetKey List	Config NetKey Add	0x8040	Client→Server
	Config NetKey Update	0x8045	Client→Server
	Config NetKey Delete	0x8041	Client→Server
	Config NetKey Status	0x8044	Server→Client
	Config NetKey Get	0x8042	Client→Server
	Config NetKey List	0x8043	Server→Client
AppKey List	Config AppKey Add	0x00	Client→Server
	Config AppKey Update	0x01	Client→Server
	Config AppKey Delete	0x8000	Client→Server
	Config AppKey Status	0x8003	Server→Client
	Config AppKey Get	0x8001	Client→Server
	Config AppKey List	0x8002	Server→Client
Model to AppKey List	Config Model App Bind	0x803D	Client→Server
	Config Model App Unbind	0x803F	Client→Server
	Config Model App Status	0x803E	Server→Client
	Config SIG Model App Get	0x804B	Client→Server
	Config SIG Model App List	0x804C	Server→Client
	Config Vendor Model App Get	0x804D	Client→Server
	Config Vendor Model App List	0x804E	Server→Client
Node Identity	Config Node Identity Get	0x8046	Client→Server
	Config Node Identity Set	0x8047	Client→Server
	Config Node Identity Status	0x8048	Server→Client
-	Config Node Reset	0x8049	Client→Server
	Config Node Reset Status	0x804A	Server→Client
Key Refresh Phase	Config Key Refresh Phase Get	0x8015	Client→Server
	Config Key Refresh Phase Set	0x8016	Client→Server
	Config Key Refresh Phase Status	0x8017	Server→Client
Heartbeat Publication	Config Heartbeat Publication Get	0x8038	Client→Server
	Config Heartbeat Publication Set	0x8039	Client→Server
	Config Heartbeat Publication Status	0x06	Server→Client
Heartbeat Subscription	Config Heartbeat Subscription Get	0x803A	Client→Server
	Config Heartbeat Subscription Set	0x803B	Client→Server
	Config Heartbeat Subscription Status	0x803C	Server→Client
Network Transmit	Config Network Transmit Get	0x8023	Client→Server
	Config Network Transmit Set	0x8024	Client→Server
	Config Network Transmit Status	0x8025	Server→Client
PollTimeout List	Config Low Power Node PollTimeout Get	0x802D	Client→Server
	Config Low Power Node PollTimeout Status	0x802E	Server→Client

コンフィグレーションステートを格納するメモリ領域は Mesh スタックに確保されます。Mesh スタックはコンフィグレーションメッセージを受信するとコンフィグレーションステートを自動的に更新するため、アプリケーションがこれらを操作する必要はありません。またアプリケーションは Mesh スタック API を使用することで各コンフィグレーションステートにアクセスできます。

2.3.3.2 ヘルスモデル

ヘルスモデルはノードの物理的な状態を監視するためのモデルです。ヘルスサーバーはノードの物理的な障害状態を保持するためのフォールトステートを持ちます。このステートは障害発生時に更新されます。またヘルスクライアントからのメッセージによって、ノードのセルフテストを実行することができます。またヘルスサーバーは人の注意を引き付ける仕組み(LED の点滅や振動など)を動作させるためのアテンションタイマステートも持ちます。この機能は、例えばプロビジョニング中のデバイスをユーザに明示する等の用途に使用します。

表 2-3 ヘルスモデルのステート

モデル名	SIG モデル ID	ステート
Health Server	0x0002	Current Fault Registered Fault Health Period Attention Timer
Health Client	0x0003	-

表 2-4 ヘルスメッセージ

ステート	メッセージ名	オペコード	方向
Current Fault	Health Current Status	0x04	Server→Client
Registered Fault	Health Fault Get	0x8031	Client→Server
	Health Fault Clear	0x802F	Client→Server
	Health Fault Clear Unacknowledged	0x8030	Client→Server
	Health Fault Status	0x05	Server→Client
	Health Fault Test	0x8032	Client→Server
	Health Fault Test Unacknowledged	0x8033	Client→Server
Health Period	Health Period Get	0x8034	Client→Server
	Health Period Set	0x8035	Client→Server
	Health Period Set Unacknowledged	0x8036	Client→Server
	Health Period Status	0x8037	Server→Client
Attention Timer	Health Attention Get	0x8004	Client→Server
	Health Attention Set	0x8005	Client→Server
	Health Attention Set Unacknowledged	0x8006	Client→Server
	Health Attention Status	0x8007	Server→Client

ヘルスステートを格納するメモリ領域は Mesh スタックに確保されます。

2.3.3.3 Generic OnOff モデル

Generic OnOff モデルは Bluetooth SIG が定義する Mesh モデルです。Generic OnOff Server は On または Off を保持する Generic OnOff ステートを持ちます。本ステートは Generic OnOff Client からのメッセージによって操作されます。

表 2-5 Generic OnOff モデルのステート

モデル名	SIG モデル ID (16 ビット)	ステート
Generic OnOff Server	0x1000	Generic OnOff (0x00: Off, 0x01: On)
Generic OnOff Client	0x1001	-

Generic OnOff ステートを保持するためのメモリ領域はアプリケーションで確保する必要があります。Mesh スタックは受信した Generic OnOff メッセージをコールバック関数で通知します。アプリケーションはコールバック関数で通知されたメッセージに従って Generic OnOff ステートを更新してください。

表 2-6 Generic OnOff メッセージ

ステート	モデル名	オペコード	方向
Generic OnOff	Generic OnOff Get	0x8201	Client→Server
	Generic OnOff Set	0x8202	Client→Server
	Generic OnOff Set Unacknowledged	0x8203	Client→Server
	Generic OnOff Status	0x8204	Server→Client

2.3.3.4 Vendor モデル

ユーザは独自の Mesh モデルを定義できます。ここでは Mesh サンプルプログラムに実装された Vendor モデルについて解説します。Mesh サンプルプログラムに実装した Vendor Server は任意の可変長データを保持する Vendor ステートを持ちます。本ステートは Vendor Client からのメッセージによって操作されます。

表 2-7 Vendor モデルのステート

モデル名	Vendor モデル ID (32 ビット)	ステート
Vendor Server	0x00010036 (デフォルト値)	Vendor state (任意の可変長データ)
Vendor Client	0x00020036 (デフォルト値)	-

表 2-8 Vendor メッセージ

ステート	メッセージ名	オペコード	方向
Vendor	Vendor Get	0xC10036 (デフォルト値)	Client→Server
	Vendor Set	0xC20036 (デフォルト値)	Client→Server
	Vendor Set Unacknowledged	0xC30036 (デフォルト値)	Client→Server
	Vendor OnOff Status	0xC40036 (デフォルト値)	Server→Client

2.4 Bluetooth Mesh スタック

Bluetooth Mesh スタックは Bluetooth Mesh Networking 仕様に準拠した多対多デバイス間で無線通信機能をアプリケーションに提供します。Mesh スタックは Flexible Software Package がライブラリ形式で提供し、Mesh スタック API によって Mesh 機能を利用することができます。

図 2-3 に Bluetooth Mesh スタックの内部構成を示します。

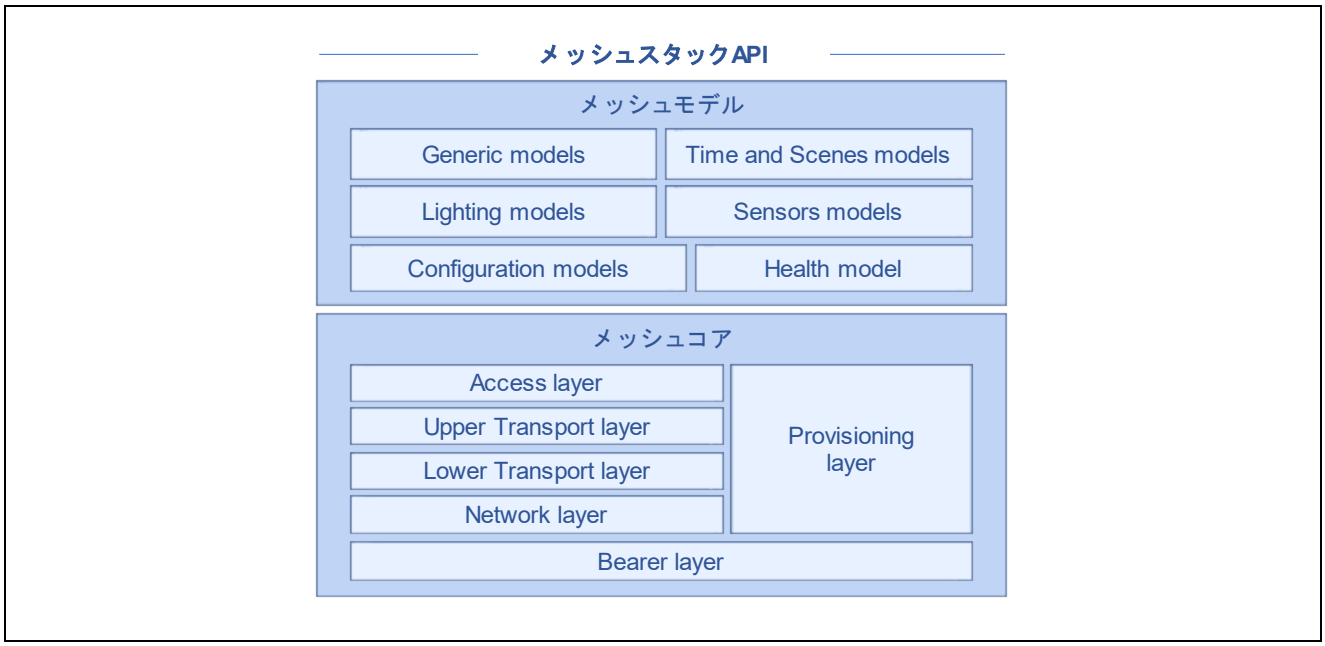

図 2-3 Bluetooth Mesh スタックの内部構成

Bluetooth Mesh スタックは次のブロックで構成されます。

- **Mesh コア**

  Mesh コアブロックは Mesh Profile 仕様で定義された各レイヤーに対応するモジュールで構成され、プロビジョニングと Mesh ネットワーク動作を実行するための機能をアプリケーションに提供します。Mesh Profile 仕様は、Specifications List の Mesh Profile 仕様書を参照してください。

- **Mesh モデル**

  Mesh モデルブロックは Mesh Model 仕様で定義された各モデルに対応するモジュールで構成され、Mesh ネットワークでの基本動作を定義した Mesh モデルをサポートするための機能をアプリケーションに提供します。Mesh Model 仕様は、Specifications List の Mesh Model 仕様書を参照してください。

Mesh スタックは Bluetooth Mesh Networking 仕様が定義するプロトコルを実現するためのモジュールで構成されます。Mesh スタック API は各モジュールに対応した下記の関数プレフィックスを持ちます。Mesh アプリケーションはアプリケーションのシナリオに応じて、Mesh スタック API を実行する必要があります。 Mesh スタック API の仕様は、「Renesas Flexible Software Package User's Manual」を参照してください。

表 2-9 Mesh スタック関数

モジュール		関数プレフィックス
Meshモデル		
	Generic OnOff	RM_MESH_GENERIC_ON_OFF_*()
	Generic Level	RM_MESH_GENERIC_LEVEL_*()
	Generic Default Transition Time	RM_MESH_GENERIC_DTT_*(),
	Generic Power OnOff	RM_MESH_GENERIC_POO_*()
	Generic Power Level	RM_MESH_GENERIC_PL_*()
	Generic Battery	RM_MESH_GENERIC_BATTERY_*()
	Generic Location	RM_MESH_GENERIC_LOC_*()
	Generic Property	RM_MESH_GENERIC_PROP_*()
	Sensor	RM_MESH_SENSOR_*()
	Time	RM_MESH_TIME_*()
	Scene	RM_MESH_SCENE_*()
	Scheduler	RM_MESH_SCHEDULER_*()
	Light Lightness	RM_MESH_LIGHT_LIGHTNESS_*()
	Light CTL	RM_MESH_LIGHT_CTL_*()
	Light HSL	RM_MESH_LIGHT_HSL_*()
	Light xyL	RM_MESH_LIGHT_XYL_*()
	Light LC	RM_MESH_LIGHT_LC_*()
	Configuration	RM_MESH_CONFIG_*()
	Health	RM_MESH_HEALTH_*()
Meshコア		
	Access Layer	RM_BLE_MESH_ACCESS_*()
	Transport Layer	RM_BLE_MESH_UPPER_TRANS_*()
	Lower Transport Layer	RM_BLE_MESH_LOWER_TRANS_*()
	Network Layer	RM_BLE_MESH_NETWORK_*()
	Bearer Layer	RM_BLE_MESH_BEARER_*()
	Provisioning Layer	RM_BLE_MESH_PROVISION_*()

2.5 Bluetooth ベアラー

Bluetooth ベアラーは、Bluetooth LE スタックのラッパー関数を Mesh スタックとアプリケーションに提供します。Bluetooth ベアラーは Flexible Software Package がライブラリ形式で提供します。Bluetooth LE スタックは、Bluetooth Low Energy 仕様に準拠した無線通信機能を上位レイヤーに提供します。Bluetooth LE スタックは Flexible Software Package がライブラリ形式で提供します。図 2-4 に Bluetooth ベアラーの内部構成を示します。メッセージ送受信のためのベアラー関数は、Mesh スタックが利用します。接続制御のためのベアラー関数は、Mesh アプリケーションが必要に応じて利用する必要があります。

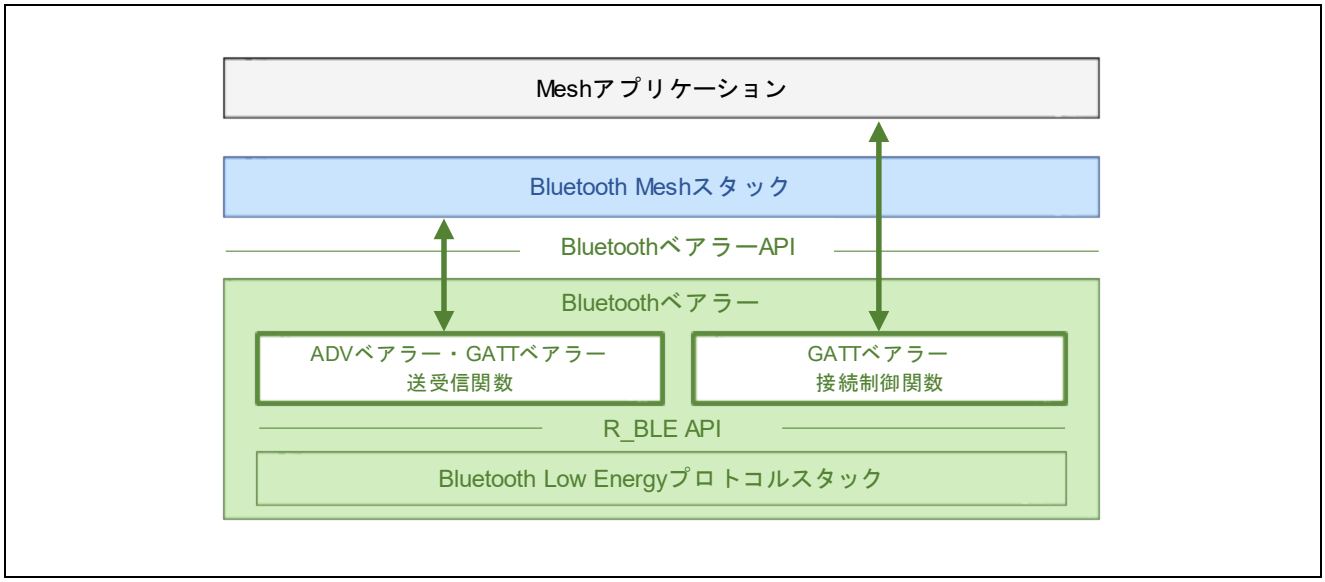

図 2-4 Bluetooth ベアラーの動作

Bluetooth ベアラーAPI 並びに R_BLE API の仕様は、「Renesas Flexible Software Package User's Manual」を参照してください。

2.5.1 メッセージ送受信のためのベアラー関数

Mesh スタックは表 2-9 に示すプレフィックスを持つメッセージ送受信関数をモデル毎に提供します。メッセージ送受信関数を利用するためには、使用前に *RM_MESH_BEARER_PLATFORM_Setup()* API を呼び出す必要があります。Mesh スタックはこれらのメッセージ送受信関数を使用して、プロビジョニング PDU および Mesh メッセージを送受信します。メッセージ送受信関数の詳細は「Renesas Flexible Software Package User's Manual」を参照してください。

2.5.2 接続制御のためのベアラー関数

Mesh スタックは GATT ベアラーのための接続状態や GATT サービスは管理しません。このため GATT ベアラーを利用する場合、Mesh アプリケーションは接続管理のためのベアラー関数を直接実行して、接続状態や GATT サービスを管理する必要があります。表 2-10 に接続制御のためのベアラー関数を示します。これらの関数は接続の確立と切断に加え、サービス探索、Notification の許可といった機能を提供します。

表 2-10 接続制御のためのベアラー関数

関数	処理	GATT Server (Peripheral)	GATT Client (Central)
RM_MESH_BEARER_PLATFORM_CallbackSet ()	GATT コールバックの登録	使用	使用
RM_MESH_BEARER_PLATFORM_SetGattMode()	GATT ベアラーモードの設定 注 1	使用	使用
RM_MESH_BEARER_PLATFORM_GetGattMode()	GATT ベアラーモードの取得 注 1	使用	使用
RM_MESH_BEARER_PLATFORM_Disconnect()	接続の切断	使用	使用
RM_MESH_BEARER_PLATFORM_SetScanResponseData()	Scan Response データの設定	使用	不使用
RM_MESH_BEARER_PLATFORM_ScanGattBearer()	接続可能なデバイスのスキャン	不使用	使用
RM_MESH_BEARER_PLATFORM_Connect()	接続の要求	不使用	使用
RM_MESH_BEARER_PLATFORM_DiscoverService()	サービスディスカバリの実行	不使用	使用
RM_MESH_BEARER_PLATFORM_ConfigureNotification()	Mesh GATT サービスの Notification 許可 注 2	不使用	使用

注 1: GATT ベアラーモードとはプロビジョニングモードまたはプロキシモード

注 2: GATT Server は Notification が許可された時点で MTU サイズを Mesh スタックに設定します。GATT Client から MTU サイズを変更する場合、GATT Client は Notification を有効化する前に MTU Exchange プロシージャを実行してください。

MTU の変更については、「RA4W1 Group Bluetooth Low Energy Application Developer's Guide」(R01AN5653)の 7.4 節を参照してください。

2.5.3 Mesh GATT サービス

GATT ベアラーによる Mesh メッセージの送受信には Mesh GATT サービスが使用されます。表 2-11 に Mesh GATT サービスの構成を示します。Mesh プロビジョニングサービスは GATT ベアラーによるプロビジョニング時に使用され、Mesh プロキシサービスはプロビジョニング完了後のプロキシ接続に使用されます。Mesh GATT サービスのどちらのサービスを公開するかは *RM_MESH_BEARER_PLATFORM_SetGattMode()* API で切り替えます。Mesh GATT サービスが定義された GATT データベースは Bluetooth ベアラーが保持しています。

表 2-11 Mesh GATT サービスの構成

サービス (UUID)	キャラクタリスティック (UUID)	プロパティ	値
Mesh Provisioning Service (0x1827)	Mesh Provisioning Data In Characteristic (0x2ADB)	Write Without Response	プロビジョニングクライアントからプロビジョニングサーバーへのプロビジョニングPDU
	Mesh Provisioning Data Out Characteristic (0x2ADC)	Notify	プロビジョニングサーバーからプロビジョニングクライアントへのプロビジョニングPDU
Mesh Proxy Service (0x1828)	Mesh Proxy Data In Characteristic (0x2ADD)	Write Without Response	プロキシクライアントからプロキシサーバーへのネットワークPDU、Meshビーコンまたはプロキシ設定を含むプロキシPDU
	Mesh Proxy Data Out Characteristic (0x2ADE)	Notify	プロキシサーバーからプロキシクライアントへのネットワークPDU、Meshビーコンまたはプロキシ設定を含むプロキシPDU

2.5.4 ADV ベアラー動作

Mesh アプリケーションが *RM_MESH_BEARER_PLATFORM_Setup()* API を実行すると、Bluetooth ベアラーは ADV ベアラーのメッセージ送受信関数を Mesh スタックに登録し、Scan を開始します。Bluetooth LE スタックが受信した Advertising パケットが Mesh スタックに通知されます。また Mesh スタックがメッセージ送信関数を実行することで、Bluetooth LE スタックは Advertising パケットを送信します。

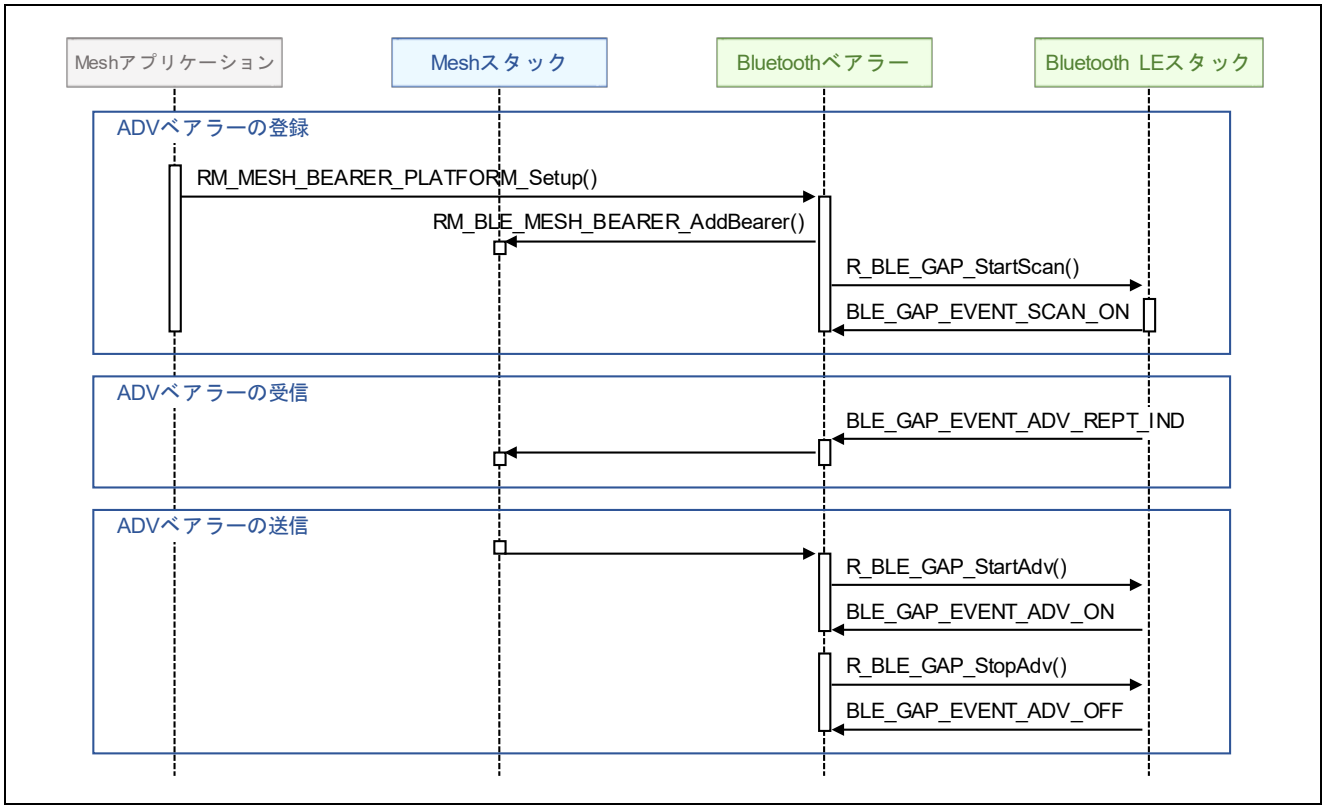

図 2-5 ADV ベアラー動作

2.5.5 GATT ベアラー動作

接続の確立後、Notification の有効化が完了すると、Bluetooth ベアラーは GATT ベアラーのメッセージ送受信関数を Mesh スタックに登録します。

GATT Server として動作する場合、Mesh スタックがメッセージ送信関数を実行すると、Bluetooth LE スタックは Notification でメッセージを送信します。また Write Without Response で受信したメッセージが Mesh スタックに通知されます。

GATT Client として動作する場合、Mesh スタックがメッセージ送信関数を実行すると、Bluetooth LE スタックは Write Without Response でメッセージを送信します。また Notification で受信したメッセージが Mesh スタックに通知されます。

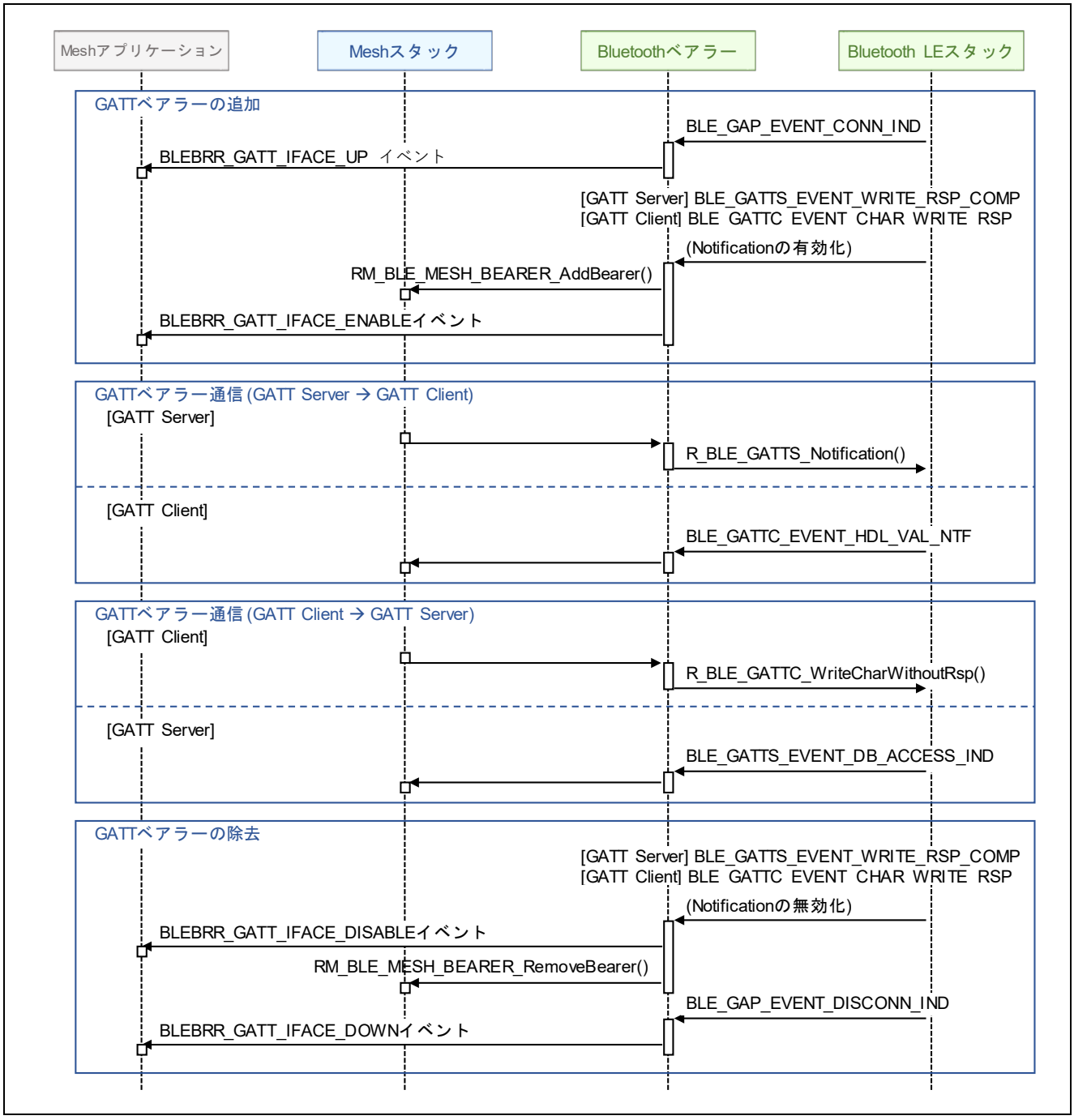

図 2-6 GATT ベアラー動作

2.6 MCU 周辺機能

Mesh サンプルプログラムは表 2-12 に示す RA4W1 周辺機能を利用します。

表 2-12 使用する RA4W1 周辺機能

RA4W1 周辺機能	FSP 周辺機能ドライバ	周辺機能を利用するソフトウェア
I/O ポート – P106、P402 および P404: EK-RA4W1 使用時 割り込みコントローラユニット(ICU) – IRQ4	r_ioport モジュール r_icu モジュール	Mesh サンプルプログラム
割り込みコントローラユニット(ICU) – BLEIRQ	r_icu モジュール	Bluetooth LE スタック
シリアルコミュニケーションインタフェース(SCI) – SCI4	r_sci_uart モジュール	Mesh サンプルプログラム
汎用 PWM タイマ (GPT) – GPT0: Bluetooth LE スタックが占有 – GPT1: Mesh スタックと Bluetooth ベアラーが共有 – GPT2: Mesh サンプルプログラムが使用	r_gpt モジュール	Mesh サンプルプログラム Mesh スタック Bluetooth ベアラー Bluetooth LE スタック
電力低減モード (LPM)	r_lpm モジュール	Mesh サンプルプログラム
データフラッシュメモリ (FLASH) – ブロック 0～5	r_flash_lp モジュール	Mesh スタック

- **I/O ポートおよび割り込みコントローラユニット(ICU)**

  Mesh サンプルプログラムは次の処理に I/O ポートを使用するために *r_ioport* モジュールおよび *r_icu* モジュールを使用します。

  - 開発ボード上の LED 制御
  - 開発ボード上のスイッチの押下検知

- **割り込みコントローラユニット(ICU)**

  Bluetooth LE スタックは BLE 割り込みを検知するために *r_icu* モジュールを使用します。

- **シリアルコミュニケーションインタフェース (SCI)**

  Mesh サンプルプログラムは UART を経由したコンソールの入出力のために *r_sci_uart* モジュールを使用します。

- **汎用 PWM タイマ (GPT)**

  BLE モジュールの Bluetooth LE スタックは GPT0 を占有します。

  Mesh スタックは GPT1 を使用して IV Update プロシージャの最小継続時間である 96 時間を監視します。また、Bluetooth ベアラーは下記の処理に GPT1 を使用します。

  - ADV ベアラーによる Advertising の送信制御

  Mesh サンプルプログラムは下記の処理に GPT2 を使用します。

  - 開発ボード上の LED 点滅
  - 開発ボード上のスイッチのチャタリング回避
  - Config Node Reset 受信時の MCU リセット遅延
  - IV Update プロシージャの終了

- **電力低減モード(LPM)**

  Mesh サンプルプログラムは MCU の消費電力低減機能を有効化するために *r_lpm* モジュールを使用します。

- **データフラッシュメモリ (FLASH)**

  データフラッシュメモリを使用するためのデータフラッシュドライバは *RM_BLE_MESH_Open*() API によって Mesh スタックに登録されます。本ドライバは *r_flash_lp* モジュールを使用してデータフラッシュにアクセスします。Mesh スタックは先頭の Block0 から r*m_ble_mesh* モジュールプロパティの */Storage/Block Number* で指定するブロック数を使用します。 よって、*rm_ble_mesh* モジュールプロパティの *Common/Data Flash Block for Security Data* プロパティで指定するブロックと *Common/Device Specific Data Flash Block* で指定するブロックは、*/Storage/Block Number* で指定する領域と重複しないように設定してください。

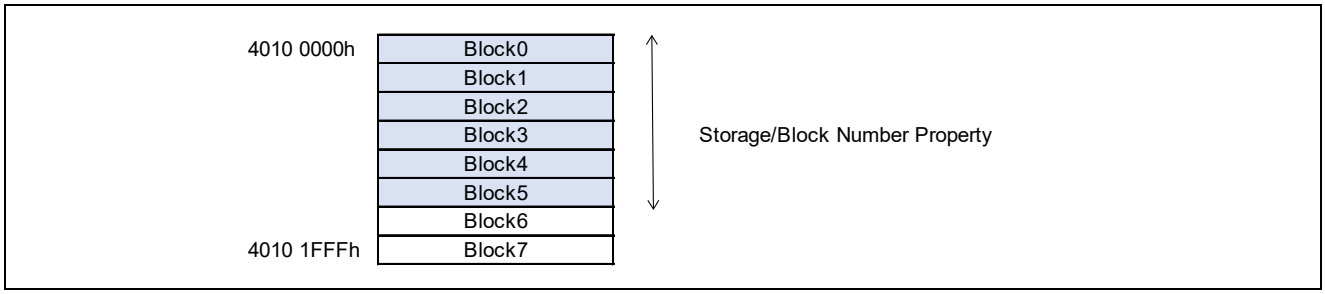

図 2-7 データフラッシュメモリの使用領域

Mesh スタックはデータフラッシュメモリに下記の情報を保持します。

- プロビジョニングデータ
  - ユニキャストアドレス
  - ネットワークキー
- コンフィグレーション情報
  - モデル構成
  - モデル設定
- IV インデックスとアップデート状態
- シーケンス番号

シーケンス番号を除く上記の情報はほとんど変更されることはありません。シーケンス番号はメッセージを送信する毎にインクリメントされます。インクリメント毎にシーケンス番号を書き込むと、データフラッシュは短期間で書き込み回数の限界に到達します。シーケンス番号をデータフラッシュに書き込む間隔を *rm_ble_mesh* モジュールの *Network Sequence Number Block Size* プロパティで指定し、データフラッシュへの書き込み頻度を低減します。

2.7 Mesh サンプルプログラムの設定

Mesh サンプルプログラムには動作を設定するための複数のコンパイルスイッチがあります。コンパイルスイッチは *mesh_appl.h* に実装されています。

```
#define IV_UPDATE_INITIATION_EN         (1)
#define LOW_POWER_FEATURE_EN            (0)
#define CONSOLE_OUT_EN                  (1)
#define ANSI_CSI_EN                     (1)
#define SCI_RCV_STRING_EN               (1)
#define SCI_RCV_STRING_BUFFER_LEN       (0x100)
```

- **IV アップデート開始処理の有効化**

  IV_UPDATE_INITIATION_EN マクロを(1)に設定することで、IV Update 開始処理が有効となります。本処理は送受信するメッセージのシーケンス番号を監視し、シーケンス番号が閾値を超えると IV Update プロシージャを実行します。これにより自ノードまたは他ノードのシーケンス番号の枯渇を防止します。

設定マクロ	設定値	内容
IV_UPDATE_INITIATION_EN	0	IV Update 開始処理を無効化
	1	IV Update 開始処理を有効化

- **ローパワー機能の有効化**

  LOW_POWER_FEATURE_EN マクロを(1)に設定することで、ローパワー機能が有効となります。プロビジョニングの完了後、フレンドノードとフレンドシップを確立し、ローパワーノードとして動作します。

設定マクロ	設定値	内容
LOW_POWER_FEATURE_EN	0	ローパワーノードへの遷移を無効化
	1	ローパワーノードへの遷移を有効化

- **コンソール出力設定**

  CONSOLE_OUT_EN マクロを(1)に設定することで、コンソールへのログ出力が有効となります。Mesh サンプルプログラムの実行する API や Mesh スタックが通知するイベントをトレースできます。

設定マクロ	設定値	内容
CONSOLE_OUT_EN	0	コンソールログ出力を無効化
	1	コンソールログ出力を有効化

- **コンソールへの ANSI CSI 出力設定**

  ANSI_CSI_EN マクロを(1)に設定することで、コンソールへの ANSI CSI (*Control Sequence Introducer*)が有効となります。Mesh サンプルプログラムは CSI によるログの色付けを行っています。使用するシリアルターミナルソフトが ANSI CSI に対応していない場合、*ANSI_CSI_EN* マクロは(0)に設定してください。

設定マクロ	設定値	内容
ANSI_CSI_EN	0	コンソールログへの ANSI CSI 出力を無効化
	1	コンソールログへの ANSI CSI 出力を有効化

- **コンソール文字列受信設定**

  SCI_RCV_STRING_EN マクロを(1)に設定することで、コンソールからの文字列受信が有効となります。受信した文字列はコールバック関数で通知されます。

設定マクロ	設定値	内容
SCI_RCV_STRING_EN	0	コンソールからの文字列受信を無効化
	1	コンソールからの文字列受信を有効化
SCI_RCV_STRING_BUFFER_LEN	0x0001～0xFFFF	文字列受信バッファサイズ

Mesh サンプルプログラムには Provisioning 動作を設定するための設定マクロがあります。設定マクロは mesh_core.c に実装されています。

```
#define CORE_PROV_BEACON_OOB_INFO           (0)
#define CORE_PROV_BEACON_URI_INFO           {.payload = "\x17//www.example.com", .length = 18}
```

- **OOB Information**

  CORE_PROV_STATIC_OOBINFO マクロには OOB Information を設定します。設定した OOB Information は Unprovisioned Device ビーコンで配布されます。複数の OOB Information を設定することができます。例えば URI とバーコードを設定する場合、(PROV_OOB_TYPE_URI | PROV_OOB_TYPE_BARCODE)を設定します。

設定マクロ	設定値	内容
CORE_PROV_BEACON_OOB_INFO	PROV_OOB_TYPE_OTHER	その他
	PROV_OOB_TYPE_URI	URI
	PROV_OOB_TYPE_2DMRC	機械判別可能な 2D コード
	PROV_OOB_TYPE_BARCODE	バーコード
	PROV_OOB_TYPE_NFC	Near Field Communication (NFC)
	PROV_OOB_TYPE_NUMBER	数値
	PROV_OOB_TYPE_STRING	文字列
	PROV_OOB_TYPE_ONBOX	ボックス上部
	PROV_OOB_TYPE_INSIDEBOX	ボックス内部
	PROV_OOB_TYPE_ONPIECEOFPAPER	紙面上
	PROV_OOB_TYPE_INSIDEMANUAL	マニュアル
	PROV_OOB_TYPE_ONDEVICE	デバイス内部

- **Encoded URI Information**

  上述の CORE_PROV_BEACON_OOB_INFO マクロに PROV_OOB_TYPE_URI を設定した場合、CORE_PROV_BEACON_URI_INFO マクロに Encoded URI Information を設定する必要があります。設定した Encoded URI Information は AD タイプの<<URI>>で配布され、Encoded URI Information の Hash 値は Unprovisioned Device ビーコンで配布されます。

設定マクロ	設定値	内容
CORE_PROV_BEACON_URI_INFO	最大 29 octets	Encoded URI URI スキームは Bluetooth SIG の Assigned Numbers で定義された"URI Scheme Name String Mapping"でエンコード

2.8 Bluetooth ベアラーの設定

Bluetooth ベアラーが実施するアドバタイジング、スキャン、接続のパラメータを以下に示します。これらの値はデバイスアドレスタイプを除き、ユーザアプリケーションから変更できません。

- **デバイスアドレスタイプ設定**

  本プロパティは *rm_mesh_bearer_platform* モジュールに属し、Bluetooth ベアラーが使用するデバイスアドレスタイプを指定できます。

プロパティ名	設定値	内容
Device Address Type	0	Public Device Address
	1	Random Device Address

- **GATT ベアラー接続のためのコネクタブルアドバタイジング設定**

パラメータ	値
アドバタイジングタイプ	Connectable and Scannable Undirected Legacy Advertising
アドバタイジングインターバル	100msec
アドバタイジングチャネルマップ	Ch37, 38, 39
フィルタポリシー	Process Scan Requests and Connection Requests from All Devices
アドバタイジングデータ長	31 bytes

- **ADV ベアラーのスキャン設定**

パラメータ	値
スキャンタイプ	Passive
スキャンインターバル	5msec
フィルタポリシー	None
デュプリケートフィルタ	None

- **GATT クライアント向けの GATT ベアラー接続設定**

パラメータ	値
コネクションインターバル	80msec
ペリフェラルレイテンシ	0
スーパービジョンタイムアウト	9.5sec

3. アプリケーション開発

本章は「RA4W1 グループ Bluetooth Mesh サンプルアプリケーション」(R01AN5848)に付属する Mesh サンプルプログラムの実装を例として、Bluetooth Mesh スタックを利用したアプリケーションの開発方法を示します。図 3-1 に Mesh サンプルプログラムの動作フローを示します。

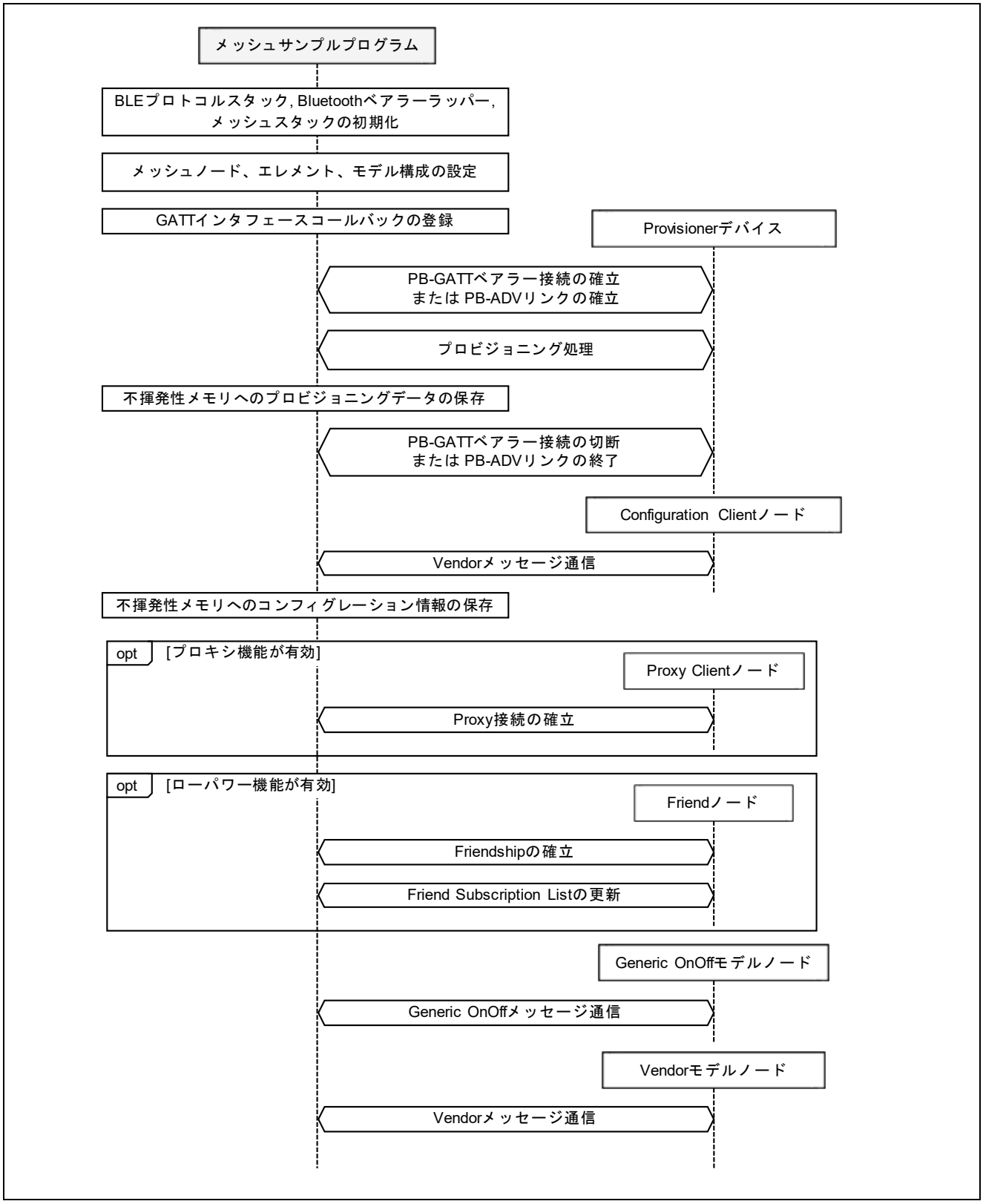

図 3-1 Mesh サンプルプログラムの動作フロー

3.1 メインルーチン

本節ではユーザアプリケーションのメインルーチンに実装すべき処理について記載します。例として挙げる Mesh サンプルプログラムでは、これらの処理は *app_main.c* に実装されています。

- **メインルーチン (app_main.c)**

アプリケーションは最初に Bluetooth LE スタックと Bluetooth ベアラーを初期化する必要があります。これらの初期化処理は Bluetooth LE スタックスケジューラによって実行されます。ベアメタル環境の Mesh サンプルプログラムでは *app_main* 関数内の while 無限ループにてスケジューラ API の *R_BLE_Execute()* API を繰り返し実行します。FreeRTOS 環境の Mesh サンプルプログラムでは同 API を繰り返し実行するタスクを生成します。初期化の完了は本節で後述するコールバック関数に通知されます。

```
void app_main(void)
{
    API_RESULT retval;
.....
    /* Initialize BLE Protocol Stack */
    R_BLE_Open();

#if (BSP_CFG_RTOS == 2)
    /* Create Semaphore */
    g_semaphore = xSemaphoreCreateBinary();

    /* Get Current Task handle */
    g_ble_core_task = xTaskGetCurrentTaskHandle();

    /* Create Execute Task */
    xTaskCreate(execute_task_entry, "execute_task", 1280, &g_ble_core_task,
                                                                    4, &g_exe_task);
#endif /* (BSP_CFG_RTOS == 2) */

.....
    /* Initialize underlying BLE Protocol Stack to use as a Mesh Bearer */
    retval = RM_MESH_BEARER_PLATFORM_Open(&g_rm_mesh_bearer_platform0_ctrl,
                                          &g_rm_mesh_bearer_platform0_cfg);
.....
```

FreeRTOS case.

R_BLE_Execute API 反復実行用タスクを生成する。

<Continue to next page>

<Continue from previous page>

```
    while (1)
    {

/* When this BLE application works on the FreeRTOS */

#if (BSP_CFG_RTOS == 2)
        if(0 != R_BLE_IsTaskFree())
        {
            vTaskSuspend(NULL);
        }
        else
        {
            xSemaphoreGive(g_semaphore);
        }
#else /* (BSP_CFG_RTOS == 2) */
        /* Process BLE Event */
        R_BLE_Execute();
#endif /* (BSP_CFG_RTOS == 2) */

.........
}

.........

#if (BSP_CFG_RTOS == 2)
void execute_task_entry(void *pvParameters)
{
    FSP_PARAMETER_NOT_USED(pvParameters);
    while(1)
    {
        xSemaphoreTake(g_semaphore, portMAX_DELAY);

        while(0 == R_BLE_IsTaskFree())
            R_BLE_Execute();

        vTaskResume(g_ble_core_task);
    }
}
.........
#endif /* (BSP_CFG_RTOS == 2) */
```

FreeRTOS case.

スケジューラに実行すべきタスクが存在する場合
R_BLE_Execute API 反復実行用タスクにセマフォを譲渡。

Baremetal case.

R_BLE_Execute API を反復実行。

FreeRTOS case.

R_BLE_Execute API を反復実行。

- Bluetooth **ベアラーの初期化完了コールバック(app_main.c)**

アプリケーションは、前述の Bluetooth ベアラーの初期化完了通知を受け取るコールバック関数を実装する必要があります。Mesh サンプルプログラムでは、同コールバック関数を *app_main.c* の *blebrr_init_cb* 関数に実装しています。本コールバック関数では *RM_BLE_MESH_Open()* API と *RM_MESH_BEARER_PLATFORM_Setup()* API を呼び出すことで Mesh スタックを初期化し Bluetooth ベアラーを Mesh スタックに登録します。これらの初期化後、Mesh アプリケーションを開始してください。Mesh アプリケーション開始にあたって呼び出す *mesh_model_config* 関数と *mesh_core_setup* 関数は、3.2 節および 3.3 節以降に説明します。

```
static void blebrr_init_cb(st_ble_dev_addr_t * own_addr)
{
……
    /* Initialize Mesh Stack */
    RM_BLE_MESH_Open(&g_ble_mesh0_ctrl, &g_ble_mesh0_cfg);

……

    /* Registers ADV Bearer with Mesh Stack and Start Scan */
    RM_MESH_BEARER_PLATFORM_Setup(&g_ble_mesh_bearer_platform0_ctrl);

……

    /* Start Mesh Application */
    mesh_model_config(&gs_mesh_model_callbacks);

    mesh_core_setup();
}
```

ノード構成を設定する。

ビーコニングを開始する。

- **Mesh スタックの終了処理**

Mesh ネットワークでの通信が不要となった場合は *RM_BLE_MESH_Close()*で Mesh スタックを終了します。

Light LC Server Model を使用した場合、*RM_MESH_LIGHT_LC_SRV_Close()*で Light LC Server Model を終了します。*RM_MESH_HEALTH_SERVER_Close()*で Health Server Model を終了後、*RM_BLE_MESH_Close()*で Mesh スタックを終了します。*RM_MESH_BEARER_PLATFORM_Close()*で Bluetooth Bearer が使用したリソースを解放します。

Bluetooth LE スタックが不要となった場合は *R_BLE_Close()*で終了します。

```
    /* Deinitialize Light LC Server Model, if it was initialized */
    RM_MESH_LIGHT_LC_SRV_Close(&g_rm_mesh_light_lc_srv0_ctrl);

    /* Deinitialize Health Server Model */
    RM_MESH_HEALTH_SERVER_Close(&g_rm_mesh_health_srv0_ctrl);

    /* Terminate Mesh Stack */
    RM_BLE_MESH_Close(&g_rm_ble_mesh0_ctrl);

    /* Free the resources allocated by Bluetooth Bearer */
    RM_MESH_BEARER_PLATFORM_Close(&g_rm_mesh_bearer_platform0_ctrl);

    /* Terminate Bluetooth LE Protocol Stack */
    R_BLE_Close();
```

3.2 ノード構成の設定

アプリケーションは、エレメントやモデルをはじめとするノード構成を設定する必要があります。この構成は、プリケーションが実行すべきシナリオによって異なります。Mesh サンプルプログラムでは、ノード構成の設定を *mesh_model.c* に実装した *mesh_model_config* 関数にて以下の様に実施しています。

```
API_RESULT mesh_model_config(const mesh_model_callbacks_t * callbacks)
{
    API_RESULT retval;

.....

    /* Create Node *//* Register Element */
    retval = g_rm_ble_mesh_access0.p_api->open(&g_rm_ble_mesh_access0_ctrl,
                                               &g_rm_ble_mesh_access0_cfg);
.....
    retval = mesh_foundation_model_register();

.....

    retval = mesh_application_model_register();


    retval = mesh_application_model_states_init();
.....
}
```

Foundation Model を登録する。

アプリケーションで使用するモデルを登録する。

アプリケーションで参照するステートを初期化する。

3.3 プロビジョニング

アプリケーションはノード構成の完了後、ネットワークに参加して他のノードと通信するため、アプリケーションはプロビジョニングサーバーとしてプロビジョニングを実行し、プロビジョニングクライアントからプロビジョニングデータを受け取る必要があります。Mesh サンプルプログラムのプロビジョニング処理は *mesh_core.c* に以下の様に実装されています。

3.3.1 プロビジョニングサーバー

(1) プロビジョニング特性とプロビジョニングコールバック関数の登録 (mesh_core.c)

mesh_core.c の *mesh_core_setup* 関数では、プロビジョニングイベント処理を実装するコールバック関数を登録し、デバイスが未プロビジョニングであった場合、認証方法等の属性を指定したビーコン送信を開始します。プロビジョニングイベント処理を実装するコールバック関数は FSP configuration において、*rm_ble_mesh_provision* モジュールの *Provision Callback* プロパティで指定します。また、*mesh_core_setup* 関数では、デバイスがプロビジョニング済であった場合の処理も実装します。同部の実装は 3.4 節を参照ください。

```
API_RESULT mesh_core_setup(void)
{
    API_RESULT retval = API_SUCCESS;
……
    retval = RM_MESH_BEARER_PLATFORM_CallbackSet(&g_rm_mesh_bearer_platform0_ctrl,
                                                 mesh_core_gatt_iface_cb);
……

    /* Check if Provisioning is not complete */
    if (API_SUCCESS != mesh_core_get_primary_unicast_address(&addr))
    {
       /* Register Provisioning capabilities */
       retval = (API_RESULT) RM_BLE_MESH_PROVISION_Open(
                                                 &g_rm_ble_mesh_provision0_ctrl,
                                                 &g_rm_ble_mesh_provision0_cfg);

       if ((FSP_SUCCESS == retval) || (FSP_ERR_ALREADY_OPEN == retval))
       {

            /* Configure as Unprovisioned Device (Provisioning Server) */
            retval = mesh_core_prov_setup(RM_BLE_MESH_PROVISION_ROLE_DEVICE,
                                          RM_BLE_MESH_PROVISION_BEARER_TYPE_ADV |
                                          RM_BLE_MESH_PROVISION_BEARER_TYPE_GATT);
       }
    }
    else
    {
        /* Configure as a Proxy Server and Start Connectable Advertising */
        mesh_core_proxy_setup ();
        mesh_core_proxy_start(RM_BLE_MESH_NETWORK_GATT_PROXY_ADV_MODE_NET_ID);
……
    }
……
}
```

プロビジョニングイベント発生時のコールバック関数を登録する。

プロビジョニング属性を指定しビーコン送信を開始する。

3.4 節参照

(2) プロビジョニングの開始 (mesh_core.c)

mesh_core.c の *mesh_core_prov_setup* 関数では、引数で指定された属性に基づいたビーコン送信を開始します。

```
static API_RESULT mesh_core_prov_setup(rm_ble_mesh_provision_role_t role,
                                       rm_ble_mesh_provision_bearer_type_t brr)
{
    API_RESULT retval;
……
    if (RM_BLE_MESH_PROVISION_BEARER_TYPE_GATT & brr)
    {
        RM_MESH_BEARER_PLATFORM_SetGattMode(&g_rm_mesh_bearer_platform0_ctrl,
                                            RM_MESH_BEARER_PLATFORM_GATT_MODE_PROVISION);
    }

……
    retval = (API_RESULT)RM_BLE_MESH_PROVISION_Setup
                (
                    &g_rm_ble_mesh_provision0_ctrl,
                    role,
                    info,
                    CORE_PROV_SETUP_TIMEOUT_SECS
                );
……
    if (API_SUCCESS == retval)
    {
        if ((RM_BLE_MESH_PROVISION_ROLE_DEVICE == role) &&
              (RM_BLE_MESH_PROVISION_BEARER_TYPE_ADV & brr))
        {
            mesh_core_prov_bind(RM_BLE_MESH_PROVISION_BEARER_TYPE_ADV,
                                &gs_prov_device);
        }
    }
……
}
```

PB-GATT を使用する場合は Mesh Provisioning Service を公開する。

ビーコン諸元を設定する。

ビーコン送信を開始する。

(3) プロビジョニングコールバック関数 (mesh_core.c)

Mesh サンプルプログラムでは、プロビジョニングイベント発生時のコールバック関数プロビジョニングイベントを受け取るためのコールバック関数を *mech_core.c* の *mesh_core_prov_cb* 関数に以下の様に実装しています。同関数において、プロビジョニングクライアントから提供されたプロビジョニングデータは *RM_BLE_MESH_PROVISION_EVENT_TYPE_PROVDATA_INFO* イベントにて *rm_ble_mesh_access* モジュールの *setProvisioningData* API を用いて Mesh スタックに登録します。

```
void mesh_core_prov_cb(rm_ble_mesh_provision_callback_args_t * p_args)
{
……

    switch (p_args->event_type)
    {
……
        case RM_BLE_MESH_PROVISION_EVENT_TYPE_PROVDATA_INFO:
            rdata = (rm_ble_mesh_provision_data_t *)(p_args->event_data.payload);
            /* Provide Provisioning Data to Access Layer */
            retval = g_rm_ble_mesh_access0.p_api->
                     setProvisioningData(&g_rm_ble_mesh_access0_ctrl, rdata);
            break;
……
    }

    return;
}
```

(4) プロビジョニングの取り消し(app_main.c)

プロビジョニングが完了するとデータフラッシュの先頭アドレス(0x40100000 番地)から 11byte にプロビジョニング済であることを表すマジックナンバーが保存されます。このマジックナンバーは *rm_ble_mesh_access* モジュールの *reset* API を用いて任意のタイミングで削除でき、プログラム再起動後にアンプロビジョンドデバイスビーコンの送信を再開することができます。「RA4W1 グループ Bluetooth Mesh サンプルアプリケーション」(R01AN5848)付属のサンプルプログラムでは、EK-RA4W1 に実装されている SW1 を押下した状態で起動すると、同マジックナンバーを消去します。

```
static void platform_reboot_timer_cb(void)
{
    API_RESULT retval;
    retval = g_rm_ble_mesh_access0.p_api->reset(&g_rm_ble_mesh_access0_ctrl);
……
}
```

3.3.2 プロビジョニングサーバーのシーケンス

(1) プロビジョニングのセットアップ

本サンプルプログラムは PB-ADV ベアラーと PB-GATT ベアラーの両方に対応しており、PB-ADV ベアラーによる Unprovisioned Device ビーコンと、PB-GATT ベアラーのための Connectable Undirected Advertising を交互に送信します。

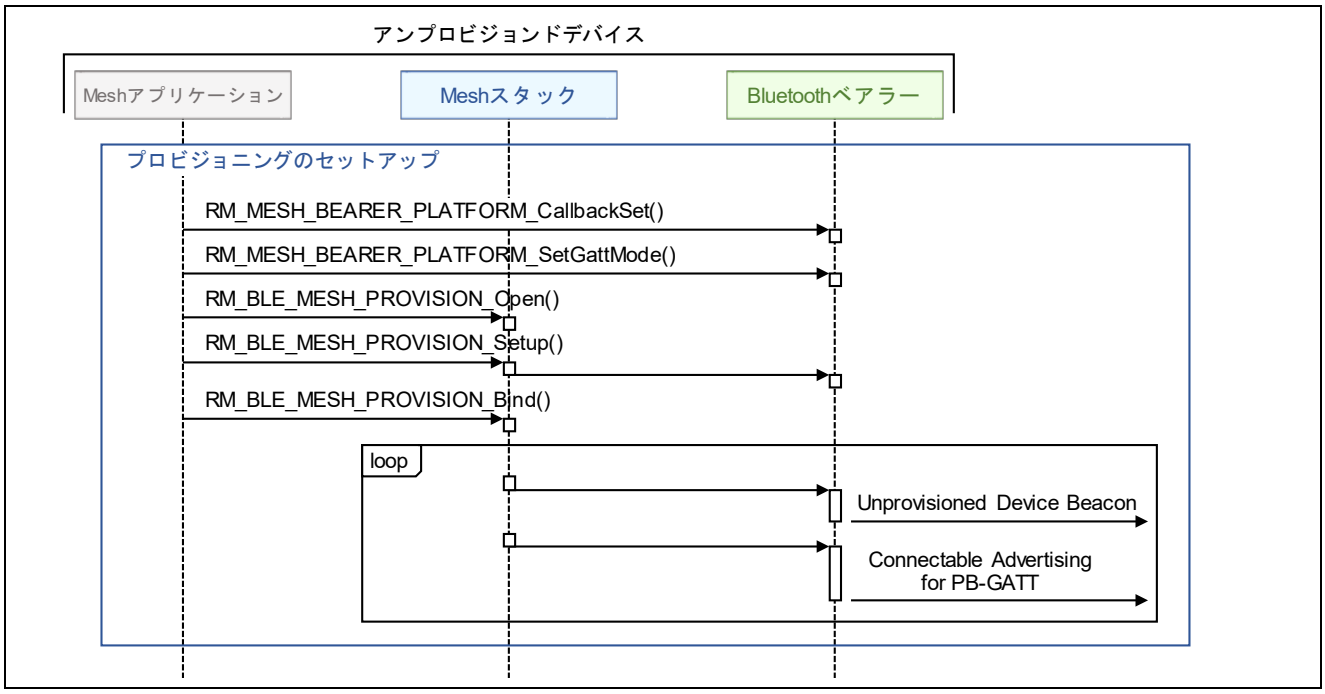

図 3-2 プロビジョニングのセットアップ

(2) PB-ADV ベアラーによるセッションの確立

PB-ADV でプロビジョニング処理を実行する場合、プロビジョニングサーバーはプロビジョニングクライアントとセッションを確立します。またプロビジョニング処理の終了後、セッションを終了します。

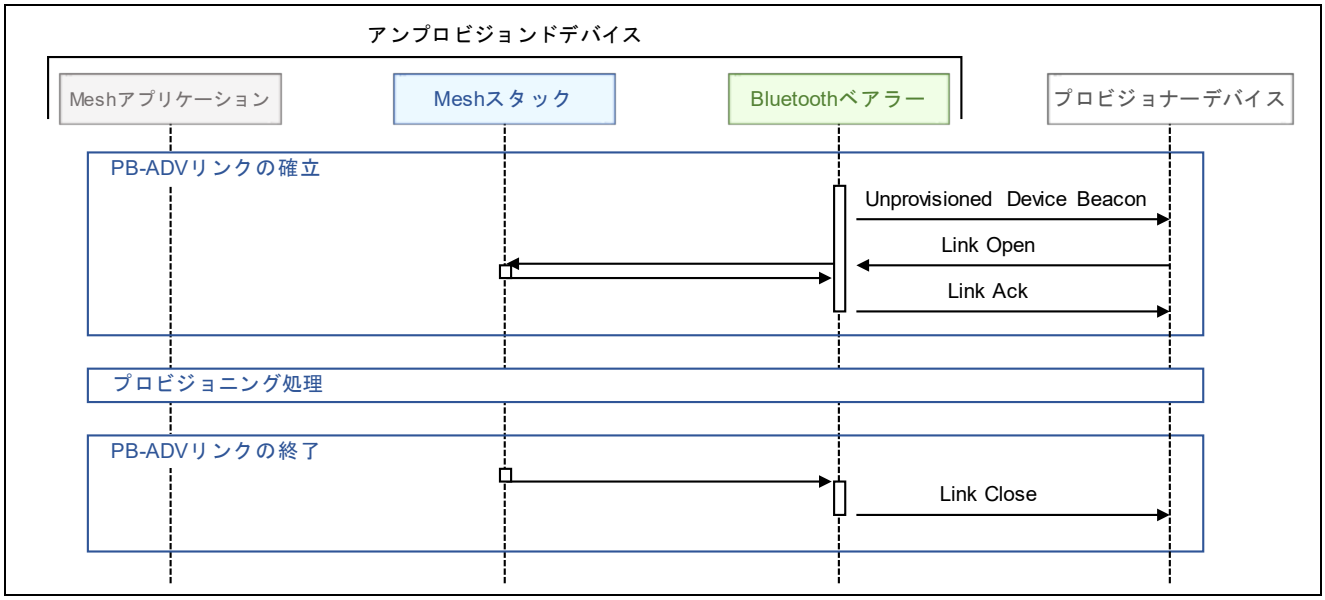

図 3-3 PB-ADV によるセッションの確立

(3) PB-GATT ベアラーによる接続の確立

PB-GATT でプロビジョニングを実行する場合、プロビジョニングクライアントはプロビジョニングサーバーと接続を確立し、Mesh プロビジョニングサービスの Notification を有効化します。またプロビジョニング処理の終了後、Notification を無効化し、接続を切断します。

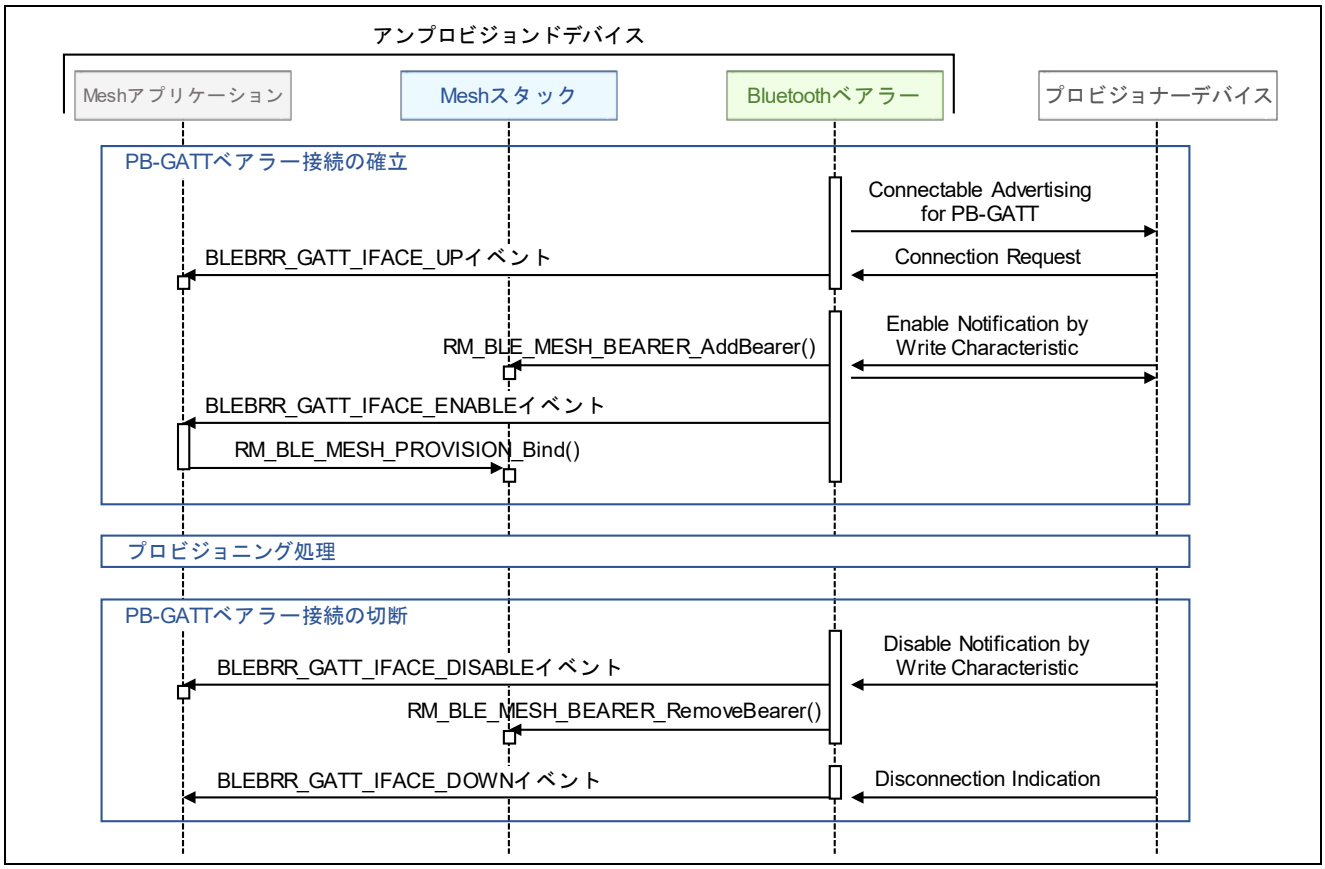

図 3-4 PB-GATT による接続の確立

(4) プロビジョニング処理

プロビジョニングベアラーによるセッションまたは接続の確立後、インビテーションからプロビジョニングデータの配布までのプロビジョニング処理が実行され、プロビジョニング PDU が交換されます。プロビジョニング処理では PB-ADV であっても PB-GATT であっても同一の処理が実行されます。

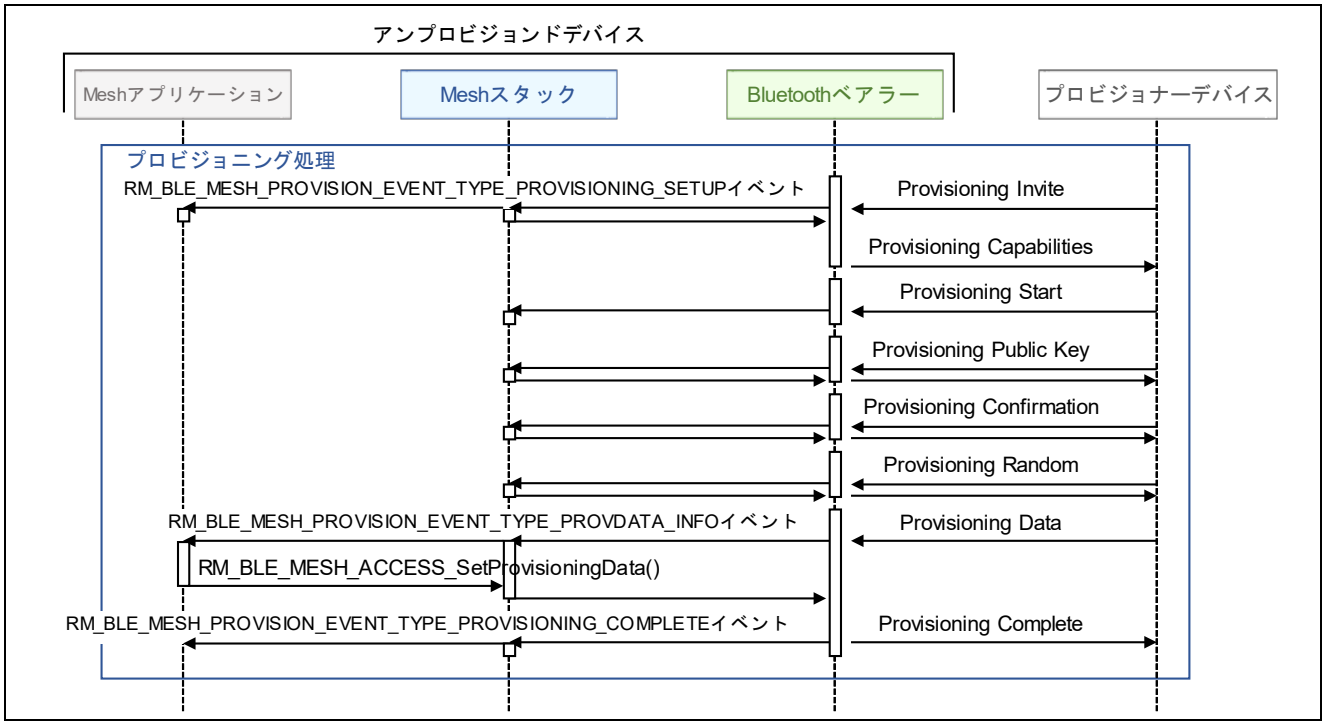

図 3-5 プロビジョニング処理

プロビジョニング処理でのセキュリティリスクの低減のため、下記が推奨されます。

- Public Key 交換ステップにおいて、OOB(Out Of Band)を使用すること。
  - OOB で Public Key を配布するには、Public Key と Private Key を *RM_BLE_MESH_PROVISION_GenerateEcdhKey*()で生成し、生成された Public Key を *RM_BLE_MESH_PROVISION_SetLocalPublicKey*()で Mesh スタックに設定します。対向のプロビジョナーデバイスへの配布方法は、RM_BLE_MESH_PROVISION_OOB_TYPE_***を *RM_BLE_MESH_PROVISION_Setup*()の引数 info.p_device->oob に設定してください。

- Authentication ステップにおいて、128 ビットで取り得る最大のエントロピーを持つ乱数値、または暗号学的にセキュアな乱数値を AuthValue として選択すること。
  - OOB Authentication の AuthValue として使用可能な 128 ビット乱数は *RM_BLE_MESH_PROVISION_GenerateRandomizedNumber*()で生成することができます。Static OOB Authentication を利用する場合、生成した AuthValue は *RM_BLE_MESH_PROVISION_SetOobAuthInfo*()で Mesh スタックに設定してください。Output OOB Authentication を利用する場合、生成した AuthValue は *RM_BLE_MESH_PROVISION_SetAuthVal*()で Mesh スタックに設定してください。

図 3-6 に OOB を使用する場合のプロビジョニング処理フローを示します。

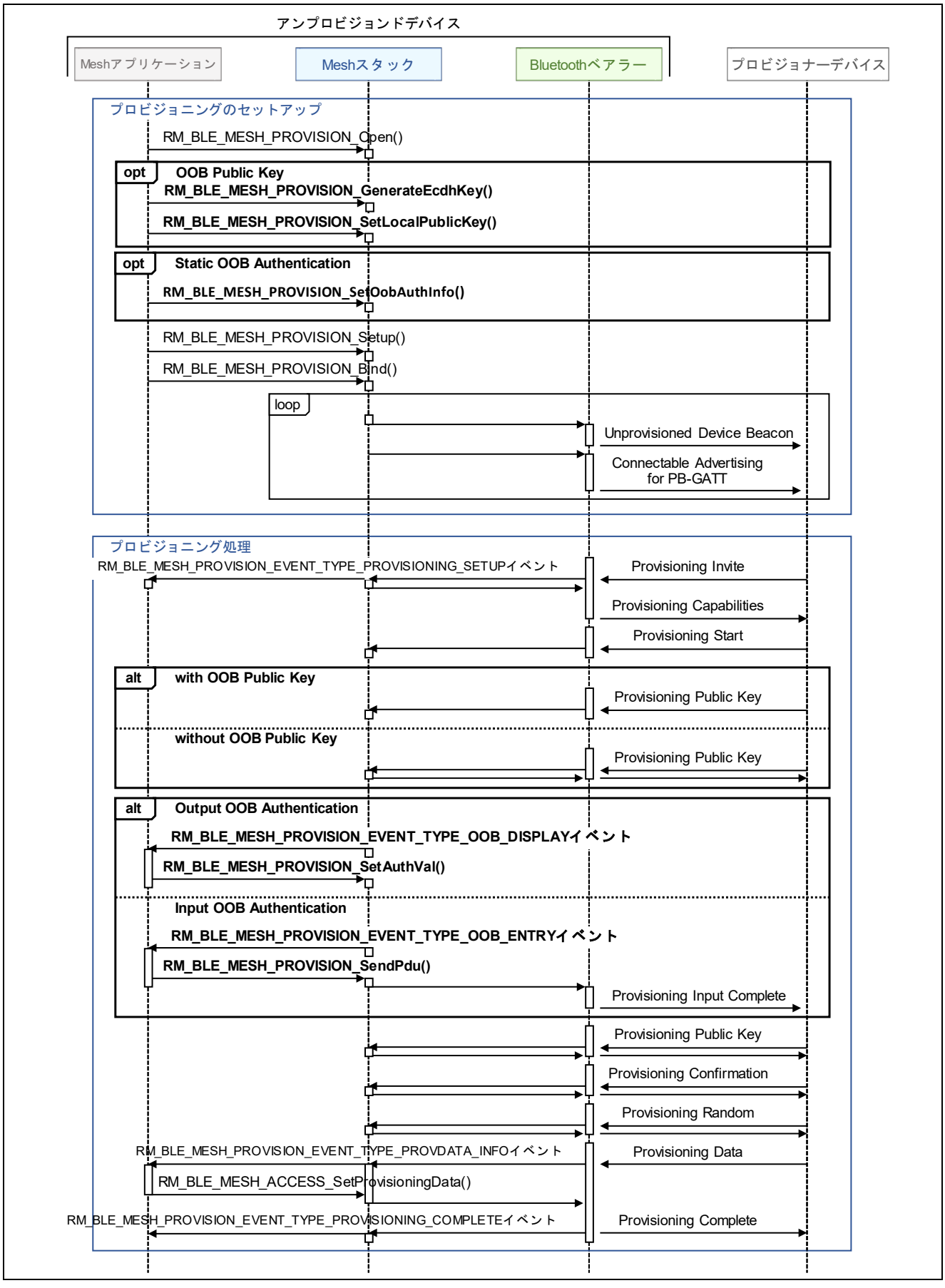

図 3-6　OOB を使用するプロビジョニング処理

3.4 プロキシ

Mesh サンプルプログラムは、プロキシサーバーまたはプロキシクライアントとして動作することができます。本節ではプロキシサーバーまたはプロキシクライアントとして動作するために必要な実装を示します。

3.4.1 プロキシサーバー

Mesh サンプルプログラムのプロキシサーバー処理を以下に示します。

(1) プロキシコールバック関数の登録 (mesh_core.c)

RM_MESH_BEARER_PLATFORM_SetGattMode() API で Bluetooth ベアラーモードを *BLEBRR_GATT_PROXY_MODE* に設定してください。また、プロキシサーバー関連のイベント処理を実装するコールバック関数は *RM_BLE_MESH_NETWORK_Open*()で Mesh スタックに登録されます。同コールバック関数は FSP configuration の *rm_ble_mesh_network* モジュールの *Callback* プロパティで指定します。Mesh サンプルプログラムにおいて、本項の処理は *mesh_core.c* の *mesh_core_proxy_setup* 関数に実装されています。

```
static API_RESULT mesh_core_proxy_setup(void)
{
    API_RESULT retval;

    RM_MESH_BEARER_PLATFORM_SetGattMode(&g_rm_mesh_bearer_platform0_ctrl,
                                        RM_MESH_BEARER_PLATFORM_GATT_MODE_PROXY);

    /* Register Proxy Callback */
    retval = RM_BLE_MESH_NETWORK_Open(&g_rm_ble_mesh_network0_ctrl,
                                      &g_rm_ble_mesh_network0_cfg);
……
    return retval;
}
```

(2) コネクタブルアドバタイジングの開始 (mesh_core.c)

プロキシクライアントとのプロキシ接続を確立するため、*RM_BLE_MESH_NETWORK_StartProxyServerAdv*() API でコネクタブルアドバタイジングを開始してください。Mesh サンプルプログラムにおいて、本処理は *mesh_core.c* の *mesh_core_proxy_start* 関数に実装されています。

```
API_RESULT mesh_core_proxy_start(uint8_t proxy_adv_mode)
{
……
    if (0 != proxy_adv_mode)
    {
        RM_BLE_MESH_NETWORK_StartProxyServerAdv(&g_rm_ble_mesh_network0_ctrl,
                                RM_BLE_MESH_NETWORK_PRIMARY_SUBNET, proxy_adv_mode);
……
    }
}
```

(3) プロキシコールバック関数 (mesh_core.c)

プロキシイベントを受け取るためのコールバック関数を実装してください。Mesh サンプルプログラムでは、*mesh_core.c* の *mesh_core_proxy_cb* 関数に実装しています。プロキシクライアントと接続を確立し、GATT プロキシサービスが有効化されると、*RM_BLE_MESH_NETWORK_EVENT_PROXY_UP* イベントが通知されます。同イベント処理部においてプロキシクライアントにキーリフレッシュフラグ、IV アップデートフラグ、カレント IV インデックスを配布するため、*RM_BLE_MESH_NETWORK_BroadcastSecureBeacon*() API で Secure Network Beacon を送信してください。

```
void mesh_core_proxy_cb(rm_ble_mesh_network_callback_args_t * p_args)
{
……
    switch (p_args->event)
    {
        case RM_BLE_MESH_NETWORK_EVENT_PROXY_UP:
……
                for (subnet_handle = 0; subnet_handle <
                      g_rm_ble_mesh0_cfg.maximum_subnets; subnet_handle++)
                {
……
                    retval =(API_RESULT)RM_BLE_MESH_NETWORK_BroadcastSecureBeacon
                                        (&g_rm_ble_mesh_network0_ctrl, subnet_handle);
……
                }
……
    }
}
```

(4) プロキシ接続の切断 (mesh_core.c)

プロキシクライアントとの接続を切断するには、*RM_MESH_BEARER_PLATFORM_Disconnect*() API をコールしてください。Mesh サンプルプログラムでは、*mesh_core.c* の *mesh_core_proxy_disconnect* 関数に実装されています。

```
API_RESULT mesh_core_proxy_disconnect(void)
{
    API_RESULT retval = API_SUCCESS;

    for (uint8_t idx = 0; idx < CORE_NUM_GATT_INTERFACES; idx++)
    {
        if (BLE_GAP_INVALID_CONN_HDL != gs_proxy_client_conn_hdl[idx])
        {
            retval = (API_SUCCESS ==
                      RM_MESH_BEARER_PLATFORM_Disconnect(&g_rm_mesh_bearer_platform
                      0_ctrl, gs_proxy_client_conn_hdl[idx])) ? retval :
                      API_FAILURE;
        }
    }

    return retval;
}
```

3.4.2 プロキシクライアント

Mesh サンプルプログラムのプロキシクライアント処理を以下に示します。

(1) プロキシコールバック関数の登録 (appl_proxy.c)

「3.4.1(1) プロキシコールバック関数の登録 (mesh_core.c)」の項を参照ください。

(2) プロキシ接続の確立 (cli_brr.c)

プロキシサーバーとのプロキシ接続を確立するため、*RM_MESH_BEARER_PLATFORM_Connect*() API をコールしてください。Mesh サンプルプログラムでは、プロキシサーバーとの接続はユーザからの *connect* コマンド入力として実装しています。*src/mesh_cli/cli/cli_brr.c* の *cli_create_gatt_conn* 関数を参照ください。

```
API_RESULT cli_create_gatt_conn(uint32_t argc, uint8_t *argv[])
{
    st_ble_dev_addr_t peer_bd_addr;
    uint8_t service_mode;
    API_RESULT retval;

……

    peer_bd_addr.type = (uint8_t)CLI_strtoi(argv[0],
                                            (uint8_t)strlen((char*)argv[0]), 16);
    CLI_strtoarray_le
    (
        argv[1],
        (uint16_t)strlen((char*)argv[1]),
        &peer_bd_addr.addr[0],
        6
    );
    service_mode = (uint8_t)CLI_strtoi(argv[2], (uint8_t)strlen((char*)argv[2]), 16);

……
    retval =    RM_MESH_BEARER_PLATFORM_Connect(&g_rm_mesh_bearer_platform0_ctrl,
                    (uint8_t*)&(peer_bd_addr.addr), peer_bd_addr.type, service_mode);

……
}
```

接続相手であるプロキシサーバーを検索するスキャン動作も同じくユーザからの *scan* コマンド入力として実装しています。発見したプロキシサーバーのデバイスアドレスは 3.3.1 節で *RM_MESH_BEARER_PLATFORM_CallbackSet* () API を用いて指定した GATT コールバック関数に *BLEBRR_GATT_IFACE_SCAN* イベントで通知されます。

(3) プロキシコールバック関数 (mesh_core.c)

プロキシイベントを受け取るためのコールバック関数を実装してください。Mesh サンプルプログラムでは、*mesh_core.c* の *mesh_core_proxy_cb* 関数に実装しています。プロキシサーバーと接続を確立し、GATT プロキシサービスが有効化されると、*RM_BLE_MESH_NETWORK_EVENT_PROXY_UP* イベントが通知されます。同イベント処理部にて *RM_BLE_MESH_NETWORK_SetProxyFilter*() API でプロキシサーバーのプロキシフィルタタイプを設定し、*rm_ble_mesh_network* モジュールの *getAllModelSubscriptionList*() API と *RM_BLE_MESH_NETWORK_ConfigProxyFilter*() API でプロキシサーバーのプロキシフィルタリストにサブスクリプションアドレスを追加してください。

```
void mesh_core_proxy_cb(ble_mesh_network_callback_args_t * p_args)
{
    switch (event)
    {
        case RM_BLE_MESH_NETWORK_EVENT_PROXY_UP:
            retval = (API_RESULT)RM_BLE_MESH_NETWORK_SetProxyFilter(NULL, &route_info,
                                                BLE_MESH_NETWORK_PROXY_FILTER_TYPE_WHITELIST);
            gs_proxy_opcode = RM_BLE_MESH_NETWORK_PROXY_CONFIG_OPECODE_SET_FILTER;
            break;

        case RM_BLE_MESH_NETWORK_EVENT_PROXY_STATUS:
            switch (gs_proxy_opcode)
            {
                case RM_BLE_MESH_NETWORK_PROXY_CONFIG_OPECODE_SET_FILTER:
                    /* Add Subscription Addresses to Proxy filter list */
                    retval = mesh_core_proxy_add_addresses(handle,
                                                           BLE_MESH_NETWORK_PRIMARY_SUBNET);
                    gs_proxy_opcode = RM_BLE_MESH_NETWORK_PROXY_CONFIG_OPECODE_ADD_TO_FILTER;
                    break;
                }
            }
            break;
    }
}

static API_RESULT mesh_core_proxy_add_addresses(NETIF_HANDLE * netif_hdl, ble_mesh_network_subnet_handle_t
                                                subnet_hdl)
{
    PROXY_ADDR addr_list[g_rm_ble_mesh0_cfg.maximum_virtual_address +
g_rm_ble_mesh0_cfg.maximum_non_virtual_address];
    UINT16     addr_count = ARRAY_SIZE(addr_list);

    retval = g_ble_mesh_access0.p_api->getAllModelSubscriptionList(&g_ble_mesh_access0_ctrl,
                                                                   &addr_count, addr_list);
    if (0 != addr_count)
    {
        ble_mesh_network_route_info_t route_info;
        route_info.interface_handle = netif_hdl;
        route_info.subnet_handle = subnet_hdl;
        ble_mesh_network_proxy_address_list_t proxy_address_list;
        proxy_address_list.address = addr_list;
        proxy_address_list.count = addr_count;
        retval = (API_RESULT)RM_BLE_MESH_NETWORK_ConfigProxyFilter(NULL, &route_info,
                                            RM_BLE_MESH_NETWORK_PROXY_CONFIG_OPECODE_ADD_TO_FILTER,
                                            &proxy_address_list);
    }

    return retval;
}
```

(4) プロキシ接続の切断 (cli_brr.c)

「3.4.1(4) プロキシ接続の切断 (mesh_core.c)」の項を参照ください。

3.4.3 プロキシのシーケンス

(1) プロキシ機能のセットアップ

Mesh サンプルプログラムはプロキシ機能に対応しており、GATT ベアラーにのみ対応するコンフィグレーションクライアントは、GATT ベアラーで本サンプルプログラムの動作を設定することができます。さらに本サンプルプログラムは、GATT ベアラーにのみ対応するノードに対して、GATT ベアラーと ADV ベアラー間でメッセージを転送することができます。

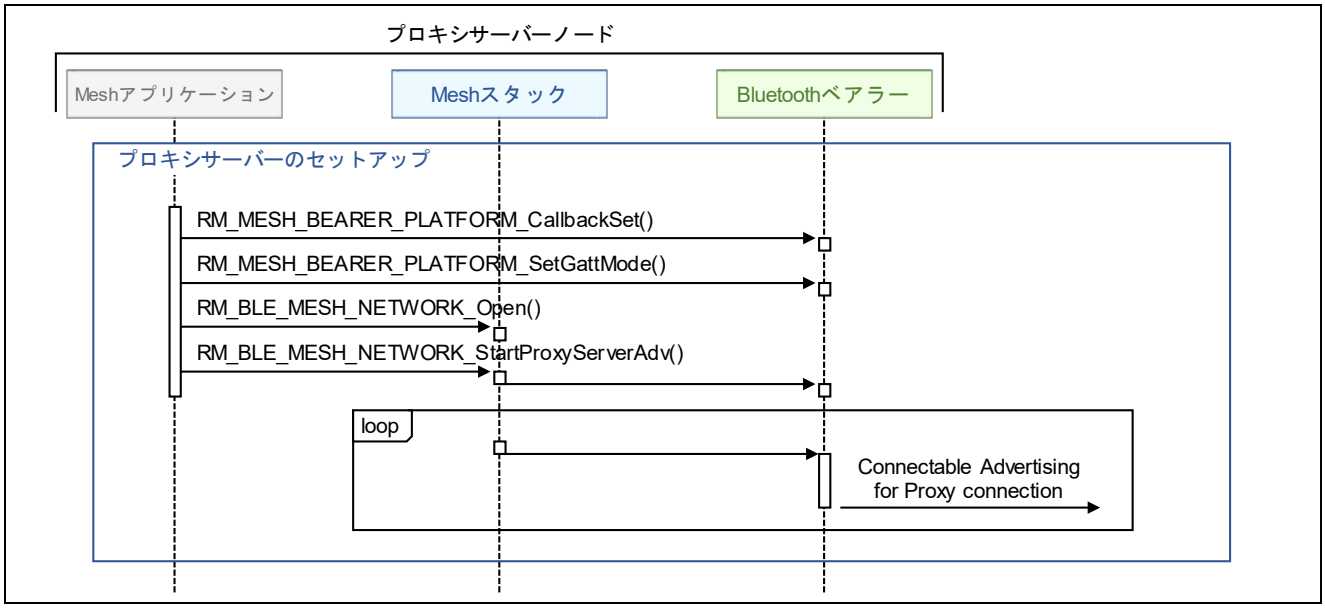

図 3-7 プロキシ機能のセットアップ

(2) プロキシ接続の確立

プロキシクライアントはプロキシサーバーと接続を確立し、Mesh プロキシサービスの Notification を有効化します。これにより、プロキシクライアントは GATT ベアラーによる Mesh メッセージの通信ができます。

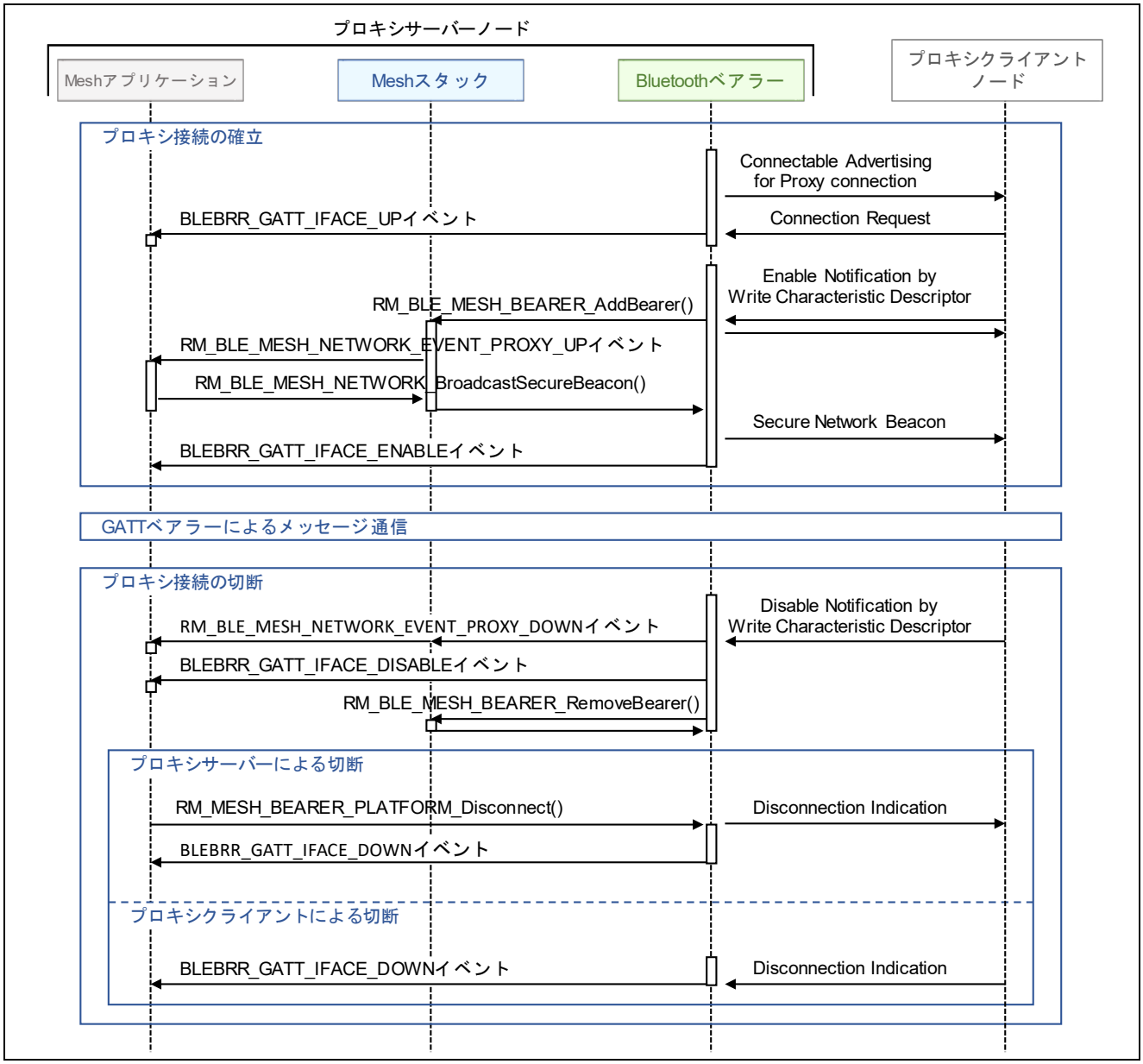

図 3-8 プロキシ接続の確立と切断

3.5 フレンドシップ

本節ではフレンドノードまたはローパワーノードとして動作するための実装を示します。

3.5.1 フレンドノード

フレンドノードとして動作するには、フレンド機能が有効化される必要があります。フレンド機能は下記のいずれかの手段によって有効化されます。

- コンフィグレーションクライアントが *Config Friend Set* メッセージを送信する。
- アプリケーションが *rm_ble_mesh_access* モジュールの *setFeaturesField* () API を実行する。

フレンド機能の有効化後、フレンドシップの確立やメッセージの保持、ローパワーノードへの応答といったフレンド機能に関連する処理は Mesh スタックが自動で実行するため、アプリケーションが対処する必要はありません。

3.5.2 ローパワーノード

ローパワーノードとして動作するには、アプリケーションがローパワー機能を有効化し、フレンドノードに対してフレンドシップ確立を要求する必要があります。フレンドシップの確立後、Mesh スタックはフレンドノードへの問い合わせと Scan の休止と再開を自動で実行します。Mesh サンプルプログラムはローパワーノードとして動作することができます。Mesh サンプルプログラムのローパワーノードとしてフレンドシップを確立するための実装を以下に示します。

注: Mesh サンプルアプリケーションにおいて、本機能はデフォルトでは無効となっています。本機能を有効化するには、2.7 節を参照して *LOW_POWER_FEATURE_EN* マクロの値を変更してください。

(1) ローパワー機能の有効化 (mesh_core.c)

3.3.1 節で解説した、プロビジョニング関連のイベント処理を担う *mesh_core.c* の *mesh_core_prov_cb* 関数にて、プロビジョニング完了時に *rm_ble_mesh_access* モジュールの *set_featureField* () API でローパワー機能を有効化してください。

```
void mesh_core_prov_cb(rm_ble_mesh_provision_callback_args_t * p_args)
{
……
    switch (p_args->event_type)
    {
……
        case RM_BLE_MESH_PROVISION_EVENT_TYPE_PROVISIONING_COMPLETE:
……

                #if LOW_POWER_FEATURE_EN
                g_rm_ble_mesh_access0.p_api->setFeaturesField
                          (&g_rm_ble_mesh_access0_ctrl, MS_ENABLE, MS_FEATURE_LPN);
                #endif /* LOW_POWER_FEATURE_EN */

                if (!gs_prov_is_gatt_iface)
                {
                  /* Configure as a Proxy Server and Start Connectable Advertising */
                    mesh_core_proxy_setup();

                    #if LOW_POWER_FEATURE_EN
                    /* Seek a Friend Node */
                    mesh_core_lpn_setup();
                    #endif /* LOW_POWER_FEATURE_EN */

……
            break;
……
    }
    return;
}
```

(2) フレンドシップ確立の要求 (mesh_core.c)

ローパワー機能有効化の後、フレンドシップコールバック関数を *RM_BLE_MESH_UPPER_TRANS_Open*() API で Mesh スタックに登録してください。同コールバック関数は FSP configuration の *rm_ble_mesh_upper_trans* モジュールの *Callback* プロパティで指定します。*RM_BLE_MESH_UPPER_TRANS_Open()* API コール後、ローパワーノードとしてフレンドシップを確立するため、*RM_BLE_MESH_UPPER_TRANS_LpnSetupFriendship*() API で *Friend Request* メッセージを送信してください。本関数の引数には、フレンドノードに対する問い合わせのタイミングに関するパラメータを設定します。

```
API_RESULT mesh_core_lpn_setup(void)
{
    API_RESULT retval;

    RM_BLE_MESH_UPPER_TRANS_Open(&g_ble_mesh_upper_trans0_ctrl, &g_ble_mesh_upper_trans0_cfg);

    ble_mesh_upper_trans_friendship_setting_t friendship_setting;
    friendship_setting.subnet_handle = 0x00;
    friendship_setting.criteria = CORE_FRIEND_CRITERIA;
    friendship_setting.rx_delay = CORE_FRIEND_RECEIVE_DELAY;
    friendship_setting.poll_timeout = CORE_FRIEND_POLLTIMEOUT;
    friendship_setting.setup_timeout = CORE_FRIEND_SETUPTIMEOUT;

    retval = RM_BLE_MESH_UPPER_TRANS_LpnSetupFriendship
                (
                    &g_ble_mesh_upper_trans0_ctrl,
                    &friendship_setting
                );

    return retval;
}
```

(3) フレンドシップコールバック関数 (mesh_core.c)

Mesh スタックが通知するフレンドシップイベントを受け取るコールバック関数を実装してください。Mesh サンプルプログラムでは、*mesh_core.c* の *mesh_core_lpn_cb* に実装しています。フレンドシップが確立すると *RM_BLE_MESH_UPPER_TRANS_EVENT_FRIENDSHIP_SETUP* イベントが通知されます。また、フレンドシップが終了すると *RM_BLE_MESH_UPPER_TRANS_EVENT_FRIENDSHIP_TERMINATE* イベントが通知されます。

ローパワーノードはフレンドノードのフレンドサブスクリプションリストにサブスクリプションアドレスを追加または除去することができます。その場合 Mesh サンプルプログラムはフレンドシップの確立後、*ble_mesh_access* モジュールの *getAllModelSubscriptionList* API でサブスクリプションリストから全てのサブスクリプションアドレスを取得し、*RM_BLE_MESH_UPPER_TRANS_LpnManageSubscription*() API でフレンドノードのフレンドサブスクリプションリストを操作します。

```
void mesh_core_lpn_cb(ble_mesh_upper_trans_callback_args_t * p_args)
{
    UCHAR  seek_req = MS_FALSE;
    UINT16 subscrn_list[g_rm_ble_mesh0_cfg.maximum_friend_subscription_list];
    UINT16 subscrn_count = g_rm_ble_mesh0_cfg.maximum_friend_subscription_list;

    switch(event_type)
    {
        case RM_BLE_MESH_UPPER_TRANS_EVENT_FRIENDSHIP_SETUP:
        {
            /* Friendship is established. */
            if (API_SUCCESS == status)
            {
                retval = g_ble_mesh_access0.p_api->getAllModelSubscriptionList
                                        (&g_ble_mesh_access0_ctrl, &subscrn_count, subscrn_list);

                if (0 != subscrn_count)
                {
                    retval = RM_BLE_MESH_UPPER_TRANS_LpnManageSubscription(NULL,
                                RM_BLE_MESH_UPPER_TRANS_CONTROL_OPCODE_FRIEND_SUBSCRN_LIST_ADD,
                                addr_list, count);
                }
            }
            /* Setup timeout is expired, and Friendship is not established. */
            else
            {
                seek_req = MS_TRUE;
            }
        }
        break;

        case RM_BLE_MESH_UPPER_TRANS_EVENT_FRIENDSHIP_TERMINATE:
        {
            /* Friendship is terminated. */
            seek_req = MS_TRUE;
        }
        break;
    }
……
}
```

3.5.3 ローパワーノードのシーケンス

(1) ローパワー機能の有効化とフレンドシップの要求

本サンプルプログラムはローパワー機能に対応しており、フレンドノードとフレンドシップを確立するための Friend Request 送信を開始します。

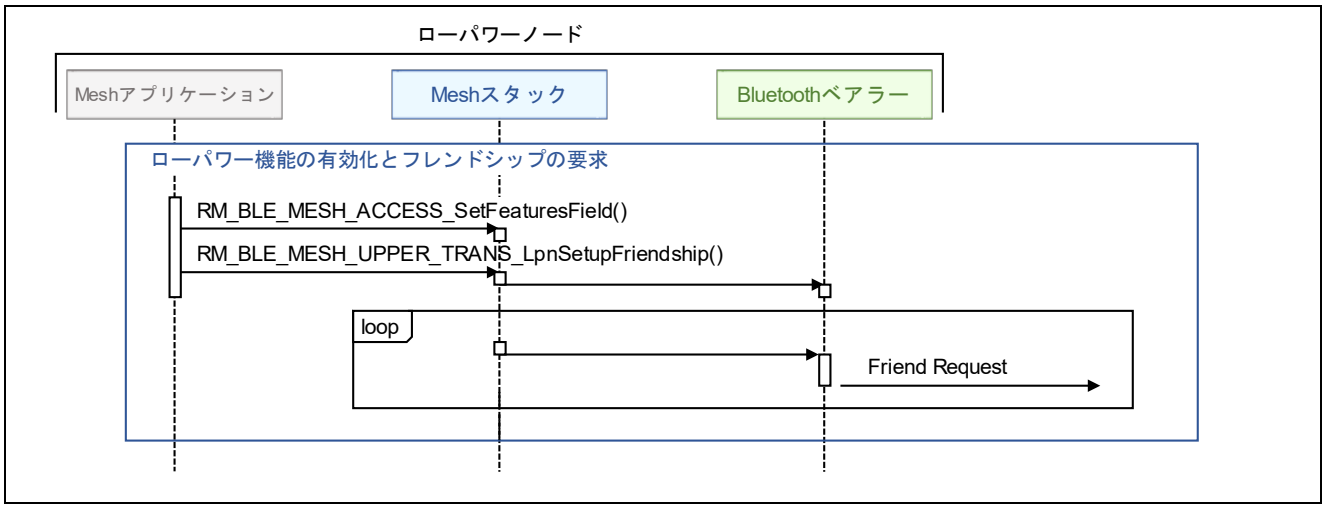

図 3-9 ローパワー機能の有効化とフレンドシップの要求

(2) フレンドシップの確立と切断

フレンドノードからの Friend Offer を受信することでフレンドシップが確立されます。フレンドシップの確立後、本サンプルプログラムはフレンドノードの Friend Subscription List に全てのサブスクリプションアドレスを登録します。これにより、これらのサブスクリプションアドレス宛のメッセージをフレンドノードが保持します。一方、Mesh スタックは定期的にメッセージポーリングを実行し、フレンドノードからメッセージを受信します。

フレンドノードへのメッセージポーリングの失敗が連続し、Friend Poll Timeout の発生によってフレンドシップが切断されると、本サンプルプログラムはフレンドシップを確立するための Friend Request 送信を再開します。

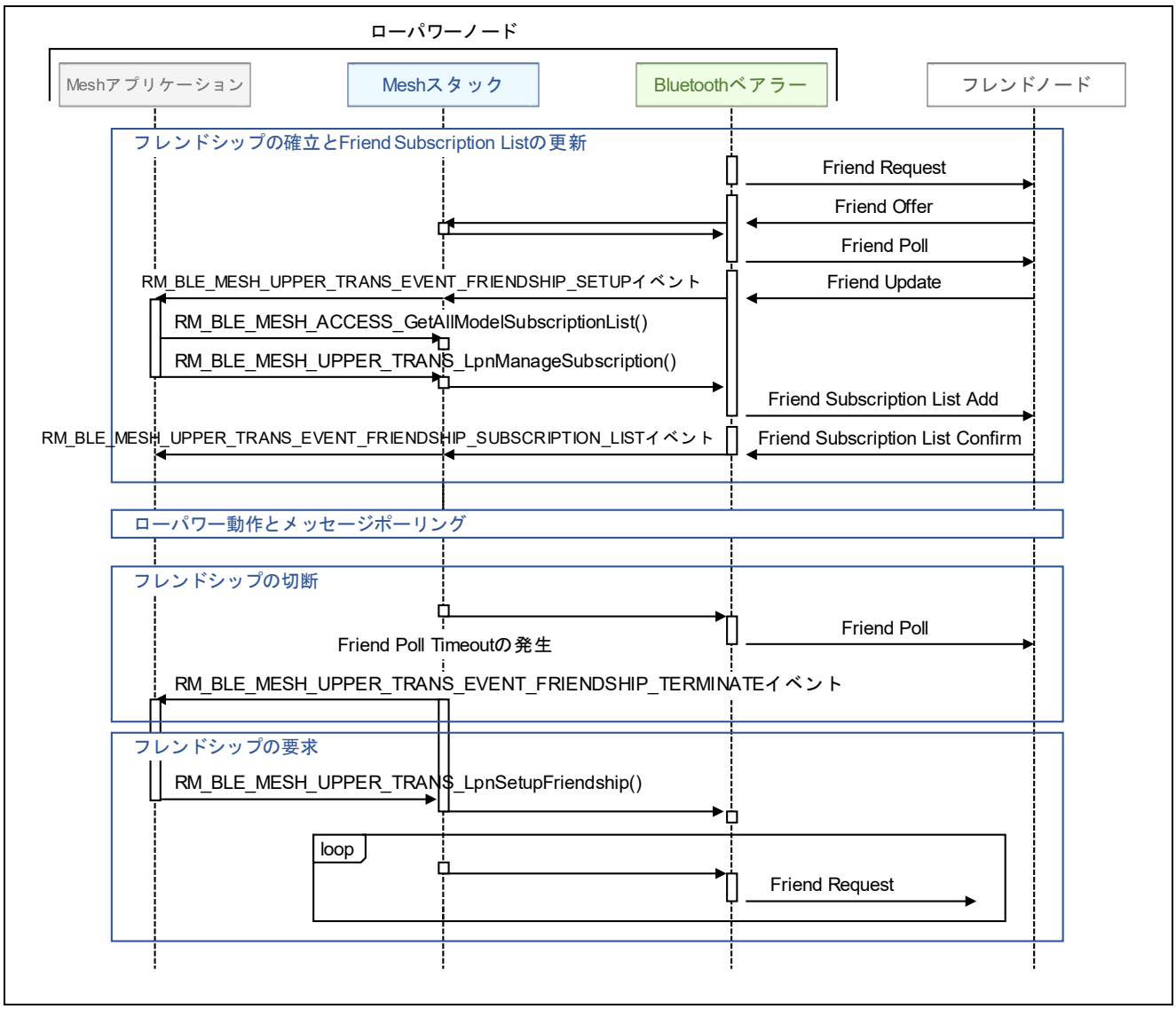

図 3-10 フレンドシップの確立と切断

ローパワーノードからフレンドシップを自発的に切断する場合は、*RM_BLE_MESH_UPPER_TRANS_LpnClearFriendship*() API で Friend Clear メッセージを送信してください。切断の完了は *RM_BLE_MESH_UPPER_TRANS_EVENT_FRIENDSHIP_CLEAR* イベントで通知されます。

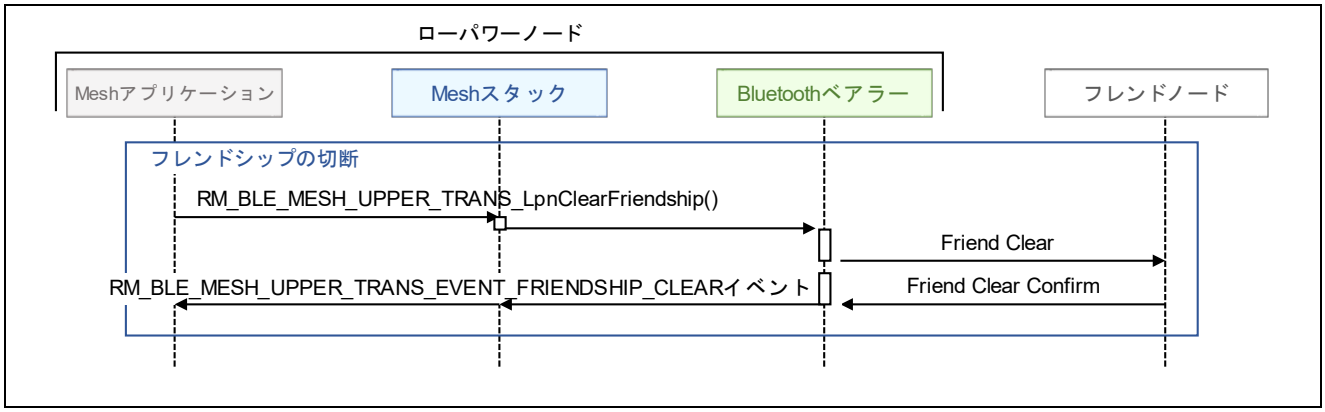

図 3-11　フレンドシップの切断

3.5.4　フレンドノードのシーケンス

(1)　フレンド機能の有効化

コンフィグレーションクライアントからフレンド機能が有効化されることで、本サンプルプログラムはフレンドノードとして動作することができます。Mesh スタックはフレンドシップの確立や切断などのフレンドノード動作に関するイベントを通知しません。

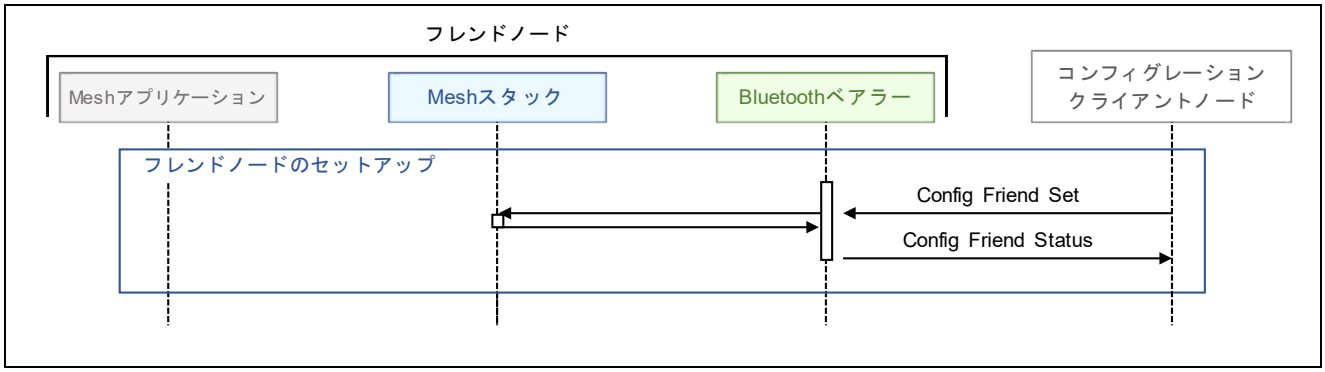

図 3-12　フレンド機能の有効化

(2) フレンドシップの確立と切断

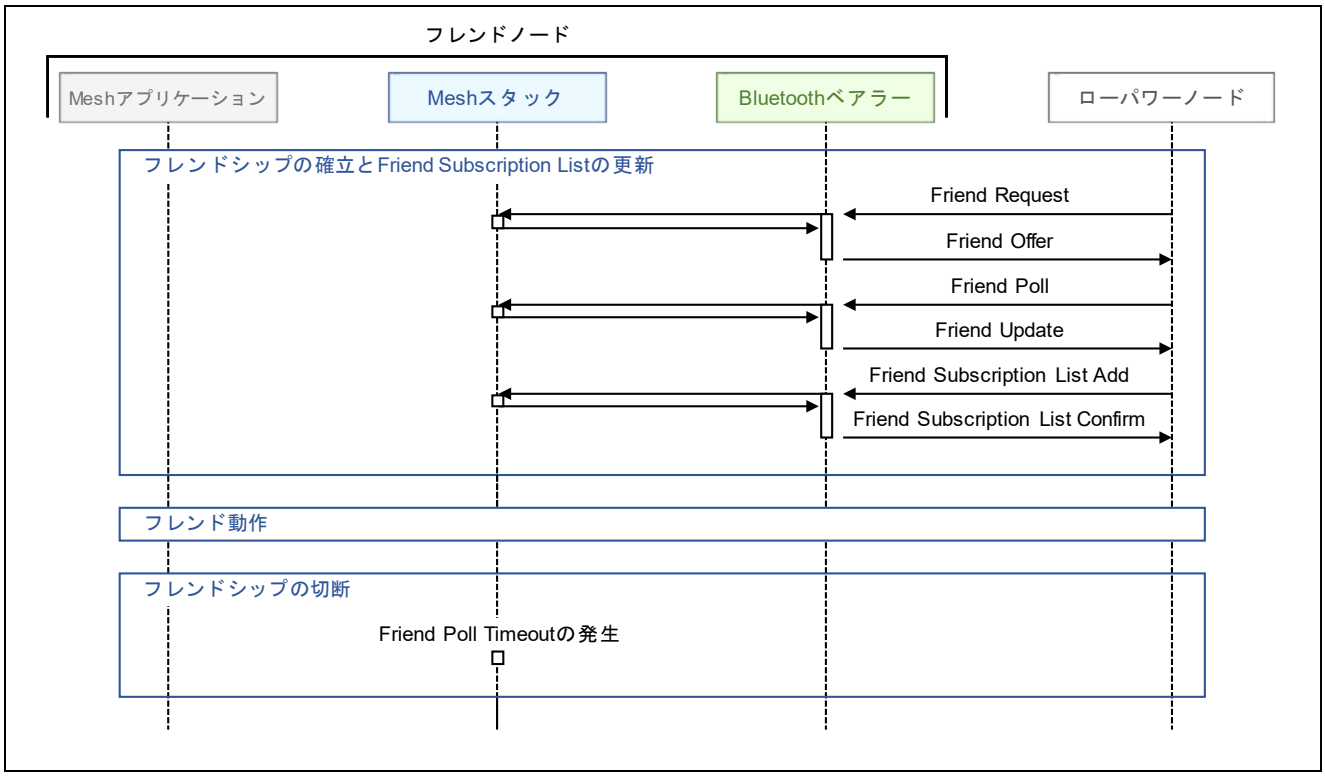

図 3-13 フレンドシップの確立と切断

3.6 コンフィグレーション

Mesh サンプルプログラムはコンフィグレーションサーバーとして動作します。本節では Mesh サンプルプログラムのコンフィグレーションサーバーモデルの実装を示します。

3.6.1 コンフィグレーションサーバー

プロビジョニングの完了後、ノードが各モデルによって通信するには、コンフィグレーションクライアントからアプリケーションキーやパブリッシュアドレス、サブスクリプションアドレスといった設定情報を受け取る必要があります。これらの設定情報は、コンフィグレーションモデルによってコンフィグレーションステートとして扱われます。

アプリケーションがコンフィグレーションサーバーモデルを登録すると、コンフィグレーションステートのメモリ領域が Mesh スタックに確保されます。Mesh スタックは、コンフィグレーションクライアントからコンフィグレーションメッセージを受信すると、コンフィグレーションステートを自動的に更新します。このため、アプリケーションがコンフィグレーションステートを管理する必要はありません。

Mesh スタックは、ローカルのコンフィグレーションステートにアクセスするための API を提供します。アプリケーションはこれらの API を使用することで、コンフィグレーションステートに直接アクセスすることもできます。コンフィグレーションステートにアクセスするための API は、「Renesas Flexible Software Package User's Manual」の *RM_BLE_MESH_ACCESS_*() API を参照してください。

(1) コンフィグレーションサーバーモデルの登録 (mesh_model.c)

RM_MESH_CONFIG_SRV_Open() API でコンフィグレーションサーバーモデルをエレメントに登録します。この処理は、Mesh サンプルプログラムでは、登録は 3.2 節で示した *mesh_model.c* に実装した *mesh_model_config* 関数から呼び出される *mesh_foundation_model_register* 関数で実施しています。

```
static API_RESULT mesh_foundation_model_register(void)
{
    API_RESULT retval;

    retval = (API_RESULT)RM_MESH_CONFIG_SRV_Open(&g_rm_mesh_config_srv0_ctrl,
                                                 &g_rm_mesh_config_srv0_cfg);

    retval = RM_MESH_HEALTH_SERVER_Open(&g_rm_mesh_health_srv0_ctrl,
                                        &g_rm_mesh_health_srv0_cfg);

    return retval;
}
```

(2) コンフィグレーションサーバーコールバック関数 (mesh_model.c)

コンフィグレーションクライアントから受信したメッセージを受け取るコールバック関数を実装します。同コールバック関数は FSP configuration における *rm_mesh_config_srv* モジュールの *Callback* プロパティにて指定します。本サンプルアプリケーションでは、*mesh_model.c* の *mesh_model_config_server_cb* 関数に実装しています。

3.6.2 コンフィグレーションサーバーのシーケンス

Config Node Reset メッセージを受信すると、Mesh スタックは全てのコンフィグレーションステートを消去します。また本サンプルプログラムは MCU をリセットし、プロビジョニングを再度実行します。

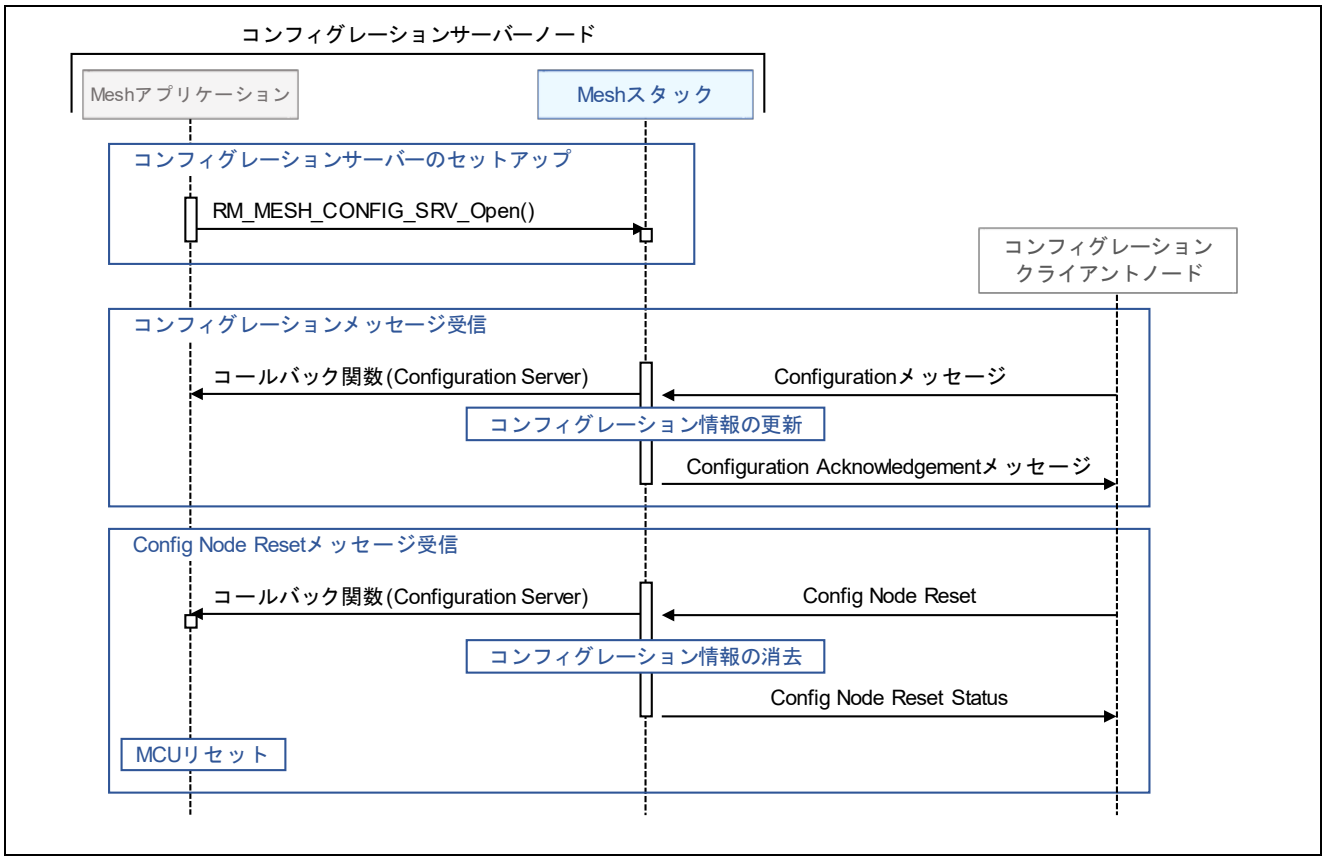

図 3-14 Mesh サンプルプログラムの Configuration Server 動作

3.7 ヘルスモデル

本節ではヘルスサーバーの実装方法を示します。

3.7.1 ヘルスサーバー

ヘルスサーバーはヘルスクライアントからの *Health Fault Test* メッセージによりセルフテストを実行します。またヘルスクライアントからの *Health Attention Set* メッセージによりアテンションタイマを起動します。

(1) ヘルスサーバーモデルの登録 (mesh_model.c)

RM_MESH_HEALTH_SERVER_Open() API でヘルスサーバーモデルをエレメントに登録します。Mesh サンプルアプリケーションでは、登録は 3.2 節で示した *mesh_model.c* に実装した *mesh_model_config* 関数から呼び出される *mesh_foundation_model_register* 関数で実施しています。

```
static API_RESULT mesh_foundation_model_register(void)
{
    API_RESULT retval;

    retval = (API_RESULT)RM_MESH_CONFIG_SRV_Open(&g_rm_mesh_config_srv0_ctrl,
                                                 &g_rm_mesh_config_srv0_cfg);

    retval = RM_MESH_HEALTH_SERVER_Open(&g_rm_mesh_health_srv0_ctrl,
                                        &g_rm_mesh_health_srv0_cfg);

    return retval;
}
```

(2) テスト関数の定義 (app_main.c)

Health Fault Test メッセージにより呼び出される関数への関数ポインタを含む構造体を app_main.c で宣言し、FSP configuration における *rm_mesh_health_srv* モジュールの *Self Test* プロパティで指定します。

```
typedef enum
{
    MESH_HEALTH_TEST_ID_00 = 0x00,
    MESH_HEALTH_TEST_ID_01 = 0x01,
    MESH_HEALTH_TEST_ID_02 = 0x02,
} e_mesh_health_test_id_t;

……

rm_ble_mesh_health_server_self_test_t gs_health_server_self_tests[] =
{
    { MESH_HEALTH_TEST_ID_00, mesh_health_self_test_00 },
    { MESH_HEALTH_TEST_ID_01, mesh_health_self_test_01 },
    { MESH_HEALTH_TEST_ID_02, mesh_health_self_test_02 },
};
```

Health Fault Test メッセージにより呼び出される関数は以下のように実装しています。同関数内でフォールトステートにヘルステスト結果を追加して *Health Fault Status* メッセージを送信するため、*RM_MESH_HEALTH_SERVER_ReportFault*() API を実行しています。

```
static void mesh_health_self_test_00(UINT8 test_id, UINT16 company_id)
{
    if ((MESH_HEALTH_TEST_ID_00 == test_id) && (g_rm_ble_mesh0_cfg.default_company_id ==
company_id))
    {
        CONSOLE_OUT("[HEALTH] A Self-Test Procedure(TestID: 0x00)\n");
        mesh_model_health_server_fault_status(MESH_HEALTH_TEST_ID_00,
                                              RM_MESH_HEALTH_SERVER_FAULT_NO_FAULT);
    }
}

……

<mesh_model.c>

API_RESULT mesh_model_health_server_fault_status(UINT8 test_id, UINT8 fault_code)
{
    API_RESULT retval;

    retval = RM_MESH_HEALTH_SERVER_ReportFault
                (
                    &g_mesh_health_srv0_ctrl,
                    &(g_mesh_health_srv0_ctrl.model_handle),
                    test_id,
                    g_rm_ble_mesh0_cfg.default_company_id,
                    fault_code
                );

    return retval;
}
```

(3) アテンションタイマコールバック関数 (app_main.c)

Health Attention Set メッセージにより実行されるアテンションタイマコールバック関数を実装します。コールバック関数は *rm_mesh_health_srv* モジュールの *Callback* プロパティで指定します。Mesh サンプルプログラムでは、*mesh_health_server_cb* として実装しています。

アテンションタイマの開始時は *RM_MESH_HEALTH_SERVER_SERVER_ATTENTION_START* イベント、再開時は *RM_MESH_HEALTH_SERVER_SERVER_ATTENTION_RESTART* イベントが通知されます。これらのイベントにより LED 点滅などのアテンション動作を開始してください。

アテンションタイマの停止時は *RM_MESH_HEALTH_SERVER_SERVER_ATTENTION_STOP* イベントが通知されます。このイベントによりアテンション動作を停止してください。

```
void mesh_health_server_cb(ble_mesh_model_health_callback_args_t * p_args)
{
    UINT8 attention_sec;

    switch (event_type)
    {
        case RM_MESH_HEALTH_SERVER_SERVER_ATTENTION_START:
        case RM_MESH_HEALTH_SERVER_SERVER_ATTENTION_RESTART:
            attention_sec = *event_param;
            if (0 != attention_sec)
            {
            }
            break;

        case RM_MESH_HEALTH_SERVER_SERVER_ATTENTION_STOP:
            break;
    }

    return API_SUCCESS;
}
```

3.7.2 ヘルスサーバーのシーケンス

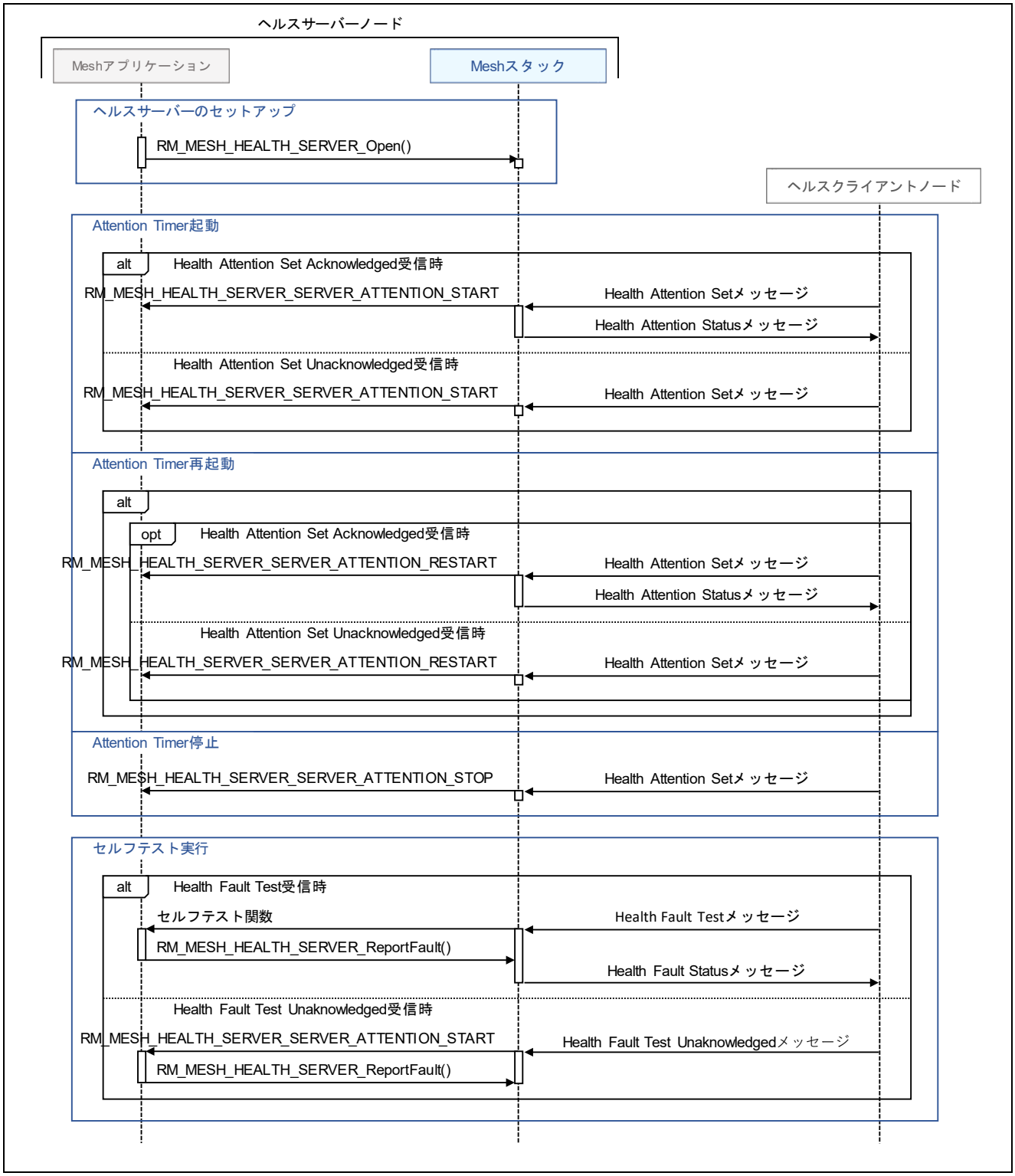

図 3-15 Mesh サンプルプログラムの Health Server 動作

3.8 アプリケーションモデル

アプリケーションが使用すべきモデルは、アプリケーションのシナリオによって異なります。アプリケーションは、一つのモデルまたは複数のモデルを使用することができます。Mesh スタックは、Bluetooth Mesh Model 仕様で定義されたモデルを使用するための API をアプリケーションに提供します。

本節では、Mesh サンプルプログラムの *Generic OnOff* モデルの実装を参照しながら、アプリケーションモデルの実装方法を示します。Mesh サンプルプログラムは *Generic OnOff* クライアントまたは *Generic OnOff* サーバーとして動作します。*Generic OnOff* クライアントは、*Generic OnOff* サーバーの *Generic OnOff* ステートを ON または OFF に変更することができます。Mesh サンプルプログラムの *Generic OnOff* モデルの実装を以下に示します。

3.8.1 サーバーモデル

(1) ステートのインスタンス (mesh_model.c)

各ステートの構造体型を使用し、ステートのインスタンスをグローバル変数として実装してください。Mesh サンプルプログラムでは、*mesh_modelc.c* にてグローバル変数で宣言しています。*Generic OnOff* モデル以外のステートの構造体型は *Renesas Flexible Software Package User's manual* を参照ください。

```
#ifdef ONOFF_SERVER_MODEL
static rm_mesh_generic_on_off_srv_info_t gs_onoff_state;
#endif /* ONOFF_SERVER_MODEL */
```

(2) ステートの初期化 (mesh_model.c)

グローバル変数として定義したステートを初期化してください。Mesh プログラムでは、3.2 節に記した *mesh_config* 関数から呼び出される *mesh_application_model_states_init* 関数にてこの初期化を実装しています。

```
    #ifdef ONOFF_SERVER_MODEL
    memset(&gs_onoff_state, 0, sizeof(gs_onoff_state));
    #endif /* ONOFF_SERVER_MODEL */
```

(3) サーバーモデルの登録 (mesh_model.c)

サーバーモデルとそのエレメントハンドル、コールバック関数の登録を実装します。コールバック関数は *rm_mesh_generic_on_off_srv* モジュールの *Callback* プロパティにて指定します。Mesh サンプルアプリケーションでは、3.2 節に記した mesh_config 関数から呼び出される *mesh_application_model_register* 関数にて、この初期化を実装しています。

```
    #ifdef ONOFF_SERVER_MODEL
    retval = RM_MESH_GENERIC_ON_OFF_SRV_Open(&g_ble_mesh_generic_on_off_srv0_ctrl,
                                             &g_ble_mesh_generic_on_off_srv0_cfg);
    #endif /* ONOFF_SERVER_MODEL */
```

(4) サーバーモデルコールバック関数 (mesh_model.c)

クライアントからのメッセージを受け取り、グローバル変数として定義されたステートを操作するためのコールバック関数を実装してください。Mesh サンプルプログラムでは、*mesh_model_onoff_server_cb* に実装されています。

```
void mesh_model_onoff_server_cb(ble_mesh_model_server_callback_args_t * p_args)
{
……

    retval = g_ble_mesh_access0.p_api->getElementHandleForModelHandle(&g_ble_mesh_access0_ctrl,
                                                    p_args->p_msg_context->handle, &elem_handle);

    if (API_SUCCESS == retval)
    {
        /* Check Message Type */
        switch (req_type->type)
        {
            case BLE_MESH_ACCESS_MODEL_REQ_MSG_TYPE_GET:
            {
                retval = mesh_model_onoff_server_state_get(state_params->state_type, &param);
……
            }
            break;

            case BLE_MESH_ACCESS_MODEL_REQ_MSG_TYPE_SET:
            {
                retval = mesh_model_onoff_server_state_set
                            (
                                state_params->state_type, state_params->state
                            );
……
            }
            break;

            default:
            break;
        }

        if (API_SUCCESS == retval)
        {
            retval = RM_MESH_GENERIC_ON_OFF_SRV_StateUpdate(&g_ble_mesh_generic_on_off_srv0_ctrl,
                                                                                    &state);
        }
    }

    return retval;
}
```

3.8.2 クライアントモデル

(1) クライアントモデルの登録 (mesh_model.c)

クライアントモデルとそのエレメントハンドル、コールバック関数の登録を実装します。コールバック関数は *rm_mesh_generic_on_off_clt* モジュールの *Callback* プロパティにて指定します。Mesh サンプルプログラムでは、3.2 節に記した *mesh_config* 関数から呼び出される *mesh_application_model_register* 関数にてこの初期化を実装しています。

```
    #ifdef ONOFF_CLIENT_MODEL
    retval = RM_MESH_GENERIC_ON_OFF_CLT_Open(&g_ble_mesh_generic_on_off_clt0_ctrl,
                                                          &g_ble_mesh_generic_on_off_clt0_cfg);
    #endif /* ONOFF_CLIENT_MODEL */
```

(2) メッセージを受け取るコールバック関数 (mesh_model.c)

サーバーからのメッセージを受け取るコールバック関数を実装します。Mesh サンプルプログラムでは、*mesh_model_onoff_client_cb* 関数に実装されています。

```
#ifdef ONOFF_CLIENT_MODEL
void mesh_model_onoff_client_cb(ble_mesh_model_client_callback_args_t * p_args)
{
    API_RESULT retval = API_SUCCESS;
    mesh_generic_on_off_status_info_t status;

    switch (opcode)
    {
        case RM_BLE_MESH_ACCESS_MESSAGE_OPCODE_GENERIC_ONOFF_STATUS:
        {
            memcpy(&status, data_param, data_len);
            status.optional_fields_present =
                    (data_len > sizeof(status.present_onoff)) ? MS_TRUE : MS_FALSE;
        }
        break;
    }

    return retval;
}
#endif /* ONOFF_CLIENT_MODEL */
```

(3) メッセージの送信関数 (mesh_model.c)

GET や *SET* といったメッセージを送信する関数を実装します。Mesh サンプルプログラムでは、これらのメソッドは *mesh_model_onoff_client_g*et, *mesh_model_onoff_client_set*, *mesh_model_onoff_client_set_unack* 関数に実装されています。これらの関数はボード上に実装したスイッチの押下やターミナルエミュレータからのコマンド入力によりコールされます。

```
#ifdef ONOFF_CLIENT_MODEL
API_RESULT mesh_model_onoff_client_get(void)
{
    API_RESULT retval;

    retval = RM_MESH_GENERIC_ON_OFF_CLT_Get(&g_ble_mesh_generic_on_off_clt0_ctrl);

    return retval;
}

API_RESULT mesh_model_onoff_client_set(UCHAR tid, UINT8 state)
{
    API_RESULT retval;
    mesh_generic_on_off_set_info_t  param;

    memset(&param, 0, sizeof(param));
    param.onoff = state;
    param.tid = tid;

    retval = RM_MESH_GENERIC_ON_OFF_CLT_Set(&g_ble_mesh_generic_on_off_clt0_ctrl, &param);

    return retval;
}

API_RESULT mesh_model_onoff_client_set_unack(UCHAR tid, UINT8 state)
{
    API_RESULT retval;
    mesh_generic_on_off_set_info_t  param;

    memset(&param, 0, sizeof(param));
    param.onoff = state;
    param.tid = tid;

    retval = RM_MESH_GENERIC_ON_OFF_CLT_SetUnacknowledged(&g_ble_mesh_generic_on_off_clt0_ctrl,
&param);

    return retval;
}
#endif /* ONOFF_CLIENT_MODEL */
```

3.8.3 Generic OnOff モデルのシーケンス

Generic OnOff Client ノードとして動作する Mesh サンプルプログラムは、ボード上のスイッチが押下されると Generic OnOff Set Unacknowledged メッセージを送信します。一方、Generic OnOff Server ノードとして動作する Mesh サンプルプログラムは、Generic OnOff Set メッセージまたは Generic OnOff Set Unacknowledged メッセージを受信すると、ボード上の LED をオンまたはオフに制御します。

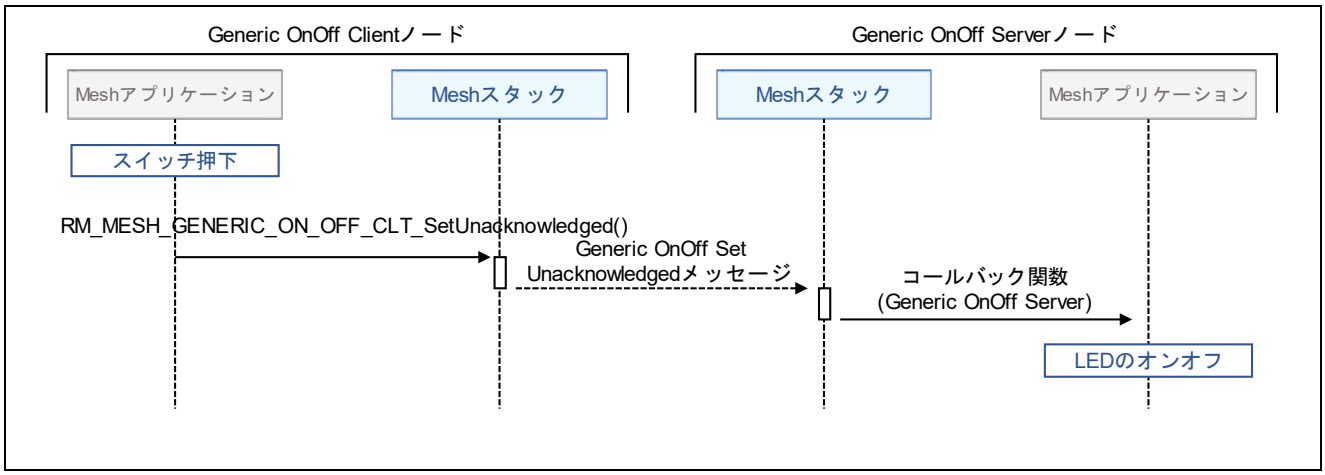

図 3-16 Mesh サンプルプログラムの Generic OnOff モデル動作

3.8.4 Vendor Model のシーケンス

Vendor Client ノードとして動作する Mesh サンプルプログラムは、コンソールから入力された文字列を Vendor Set Unacknowledged メッセージで送信します。一方、Vendor Server ノードとして動作するサンプルプログラムは、Vendor Set メッセージまたは Vendor Set Unacknowledged メッセージを受信すると、コンソールに文字列を出力します。

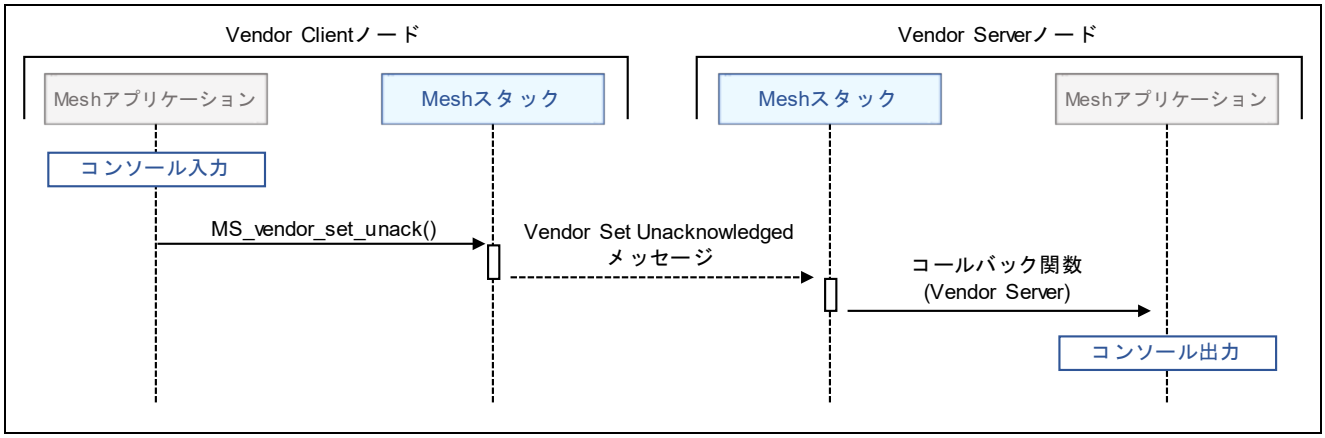

図 3-17 Mesh サンプルプログラムの Vendor モデル動作

4. Appendix

4.1 Command Line Interface プログラム

Command Line Interface (CLI)は、PC からシリアルインタフェース経由で Mesh スタック API を実行するためのインタフェースです。「RA4W1 グループ Bluetooth Mesh サンプルアプリケーション」(R01AN5848)には Command Line Interface プログラム(mesh_cli)が含まれます。本プログラムを使用することで、Mesh スタックの無線通信動作を確認できます。さらに、Mesh スタック API を使用例として、本プログラムのソースコードを参照することができます。図 4-1 に Command Line Interface プログラムの使用例を示します。

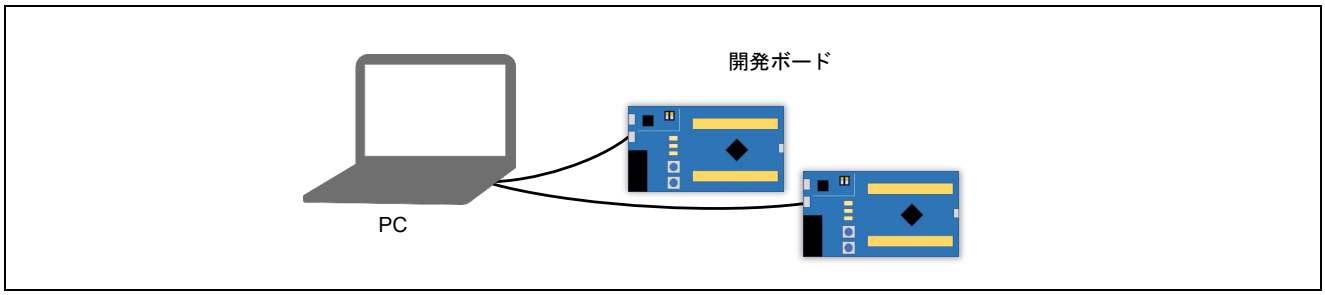

図 4-1 Command Line Interface プログラムの使用例

図 4-2 に Command Line Interface のシーケンス例を示します。本プログラムは Provisioning Client と Provisioning Server、Configuration Client と Configuration Server といった両方のロールを実行できます。

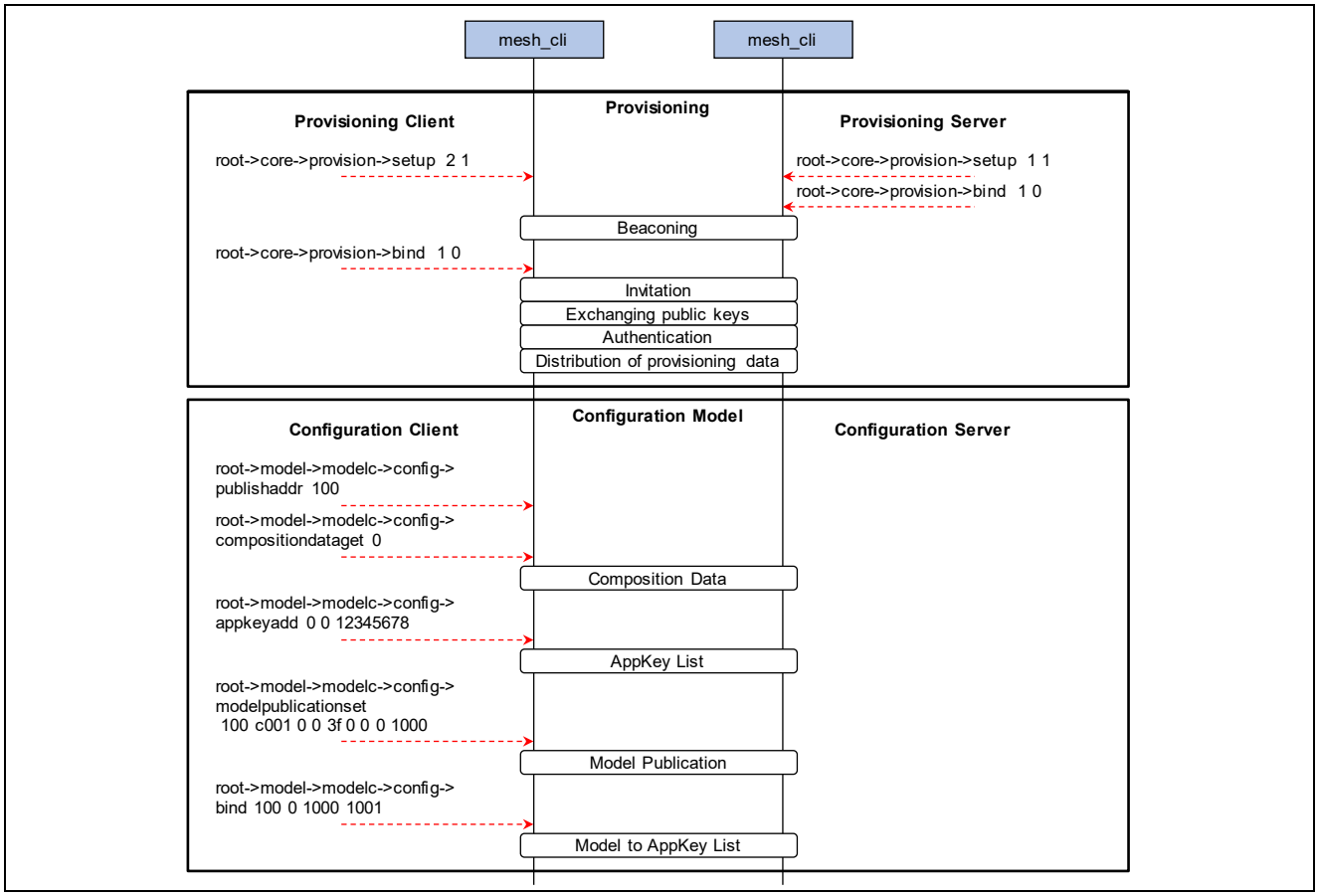

図 4-2 Command Line Interface のシーケンス例

Command Line Interface プログラムのビルド環境構築については、「RA4W1 グループ Bluetooth Mesh サンプルアプリケーション アプリケーションノート」(R01AN5848)の 2 章を参照し、ワークスペースディレクトリに生成された *mesh_cli* プロジェクトを使用してください。

EK-RA4W1 は、PC と通信するための USB シリアル変換を搭載しています。Command Line Interface を操作するには、PC 上でシリアルターミナルツールを使用してください。(例：Tera Term)

表 4-1 に Command Line Interface プログラムと通信するためのシリアルポート設定を示します。

表 4-1　シリアルポート設定

項目	設定
ボーレート	115200 bps
データ	8 bits
パリティ	なし
ストップ	1 bit
フロー制御	なし

Command Line Interface の仕様については、「RA4W1 グループ Bluetooth Mesh サンプルアプリケーション アプリケーションノート」(R01AN5848)に同梱された *mesh_cli_guide.pdf* を参照してください。

4.2　プログラムサイズ

FSP により静的ライブラリとして提供される Mesh スタック、Bluetooth LE スタック、本文 2.3.1 節並びに 2.3.2 節で説明し、「RA4W1 グループ Bluetooth Mesh サンプルアプリケーション アプリケーションノート」(R01AN5848)に付属する Mesh コアモジュールと Mesh モデルモジュールが要するプログラムサイズは以下の通りです。

分類	ROM サイズ	RAM サイズ
Mesh コアモジュール	4.77KB	0.03KB
Mesh モデルモジュール	3.61KB	0.12KB
Bluetooth LE スタック＋Mesh スタック*1 *2	304.81KB	46.56KB

*1Generic ON/OFF モデル、Configuration モデル、Health モデル含む値です。

*2 Bluetooth LE スタックは Extended Configuration です。

上記の表には、ユーザアプリケーション、ユーザアプリケーションが使用する MCU 周辺モジュール、FreeRTOS カーネルおよび BSP は含まれていません。

商標権および著作権

Bluetooth® のワードマークおよびロゴは Bluetooth SIG, Inc が所有する登録商標であり、ルネサスエレクトロニクス株式会社はこれらのマークをライセンスに基づいて使用しています。その他の商標および登録商標はそれぞれの所有者に帰属します。

RA4W1 グループ Bluetooth Mesh スタックは次のオープンソースソフトウェアを使用します。

crackle; AES-CCM, AES-128bit 機能

改訂記録

Rev.	発行日	改定内容	
1.00	2022.02.25	-	初版発行
1.01	2022.04.27	-	3.3.1 節(4)項追加
1.05	2022.12.27	P.35 P.36 P.37 P.41 P.48	「コンソール文字列受信設定」の解説を追加 Mesh サンプルプログラムの Provisioning 動作設定の解説を追加 GATT ベアラー接続のためのアドバタイジングインターバルを修正 「Mesh スタックの終了処理」の解説を追加 OOB Public Key と OOB Authentication の使用方法の解説を更新

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 静電気対策
   CMOS 製品の取り扱いの際は静電気防止を心がけてください。CMOS 製品は強い静電気によってゲート絶縁破壊を生じることがあります。運搬や保存の際には、当社が出荷梱包に使用している導電性のトレーやマガジンケース、導電性の緩衝材、金属ケースなどを利用し、組み立て工程にはアースを施してください。プラスチック板上に放置したり、端子を触ったりしないでください。また、CMOS 製品を実装したボードについても同様の扱いをしてください。
2. 電源投入時の処置
   電源投入時は、製品の状態は不定です。電源投入時には、LSI の内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。
3. 電源オフ時における入力信号
   当該製品の電源がオフ状態のときに、入力信号や入出力プルアップ電源を入れないでください。入力信号や入出力プルアップ電源からの電流注入により、誤動作を引き起こしたり、異常電流が流れ内部素子を劣化させたりする場合があります。資料中に「電源オフ時における入力信号」についての記載のある製品は、その内容を守ってください。
4. 未使用端子の処理
   未使用端子は、「未使用端子の処理」に従って処理してください。CMOS 製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI 周辺のノイズが印加され、LSI 内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。
5. クロックについて
   リセット時は、クロックが安定した後、リセットを解除してください。プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。
6. 入力端子の印加波形
   入力ノイズや反射波による波形歪みは誤動作の原因になりますので注意してください。CMOS 製品の入力がノイズなどに起因して、V_{IL}（Max.）から V_{IH}（Min.）までの領域にとどまるような場合は、誤動作を引き起こす恐れがあります。入力レベルが固定の場合はもちろん、V_{IL}（Max.）から V_{IH}（Min.）までの領域を通過する遷移期間中にチャタリングノイズなどが入らないように使用してください。
7. リザーブアドレス（予約領域）のアクセス禁止
   リザーブアドレス（予約領域）のアクセスを禁止します。アドレス領域には、将来の拡張機能用に割り付けられている リザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。
8. 製品間の相違について
   型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。同じグループのマイコンでも型名が違うと、フラッシュメモリ、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ幅射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。回路、ソフトウェアおよびこれらに関連する情報を使用する場合、お客様の責任において、お客様の機器・システムを設計ください。これらの使用に起因して生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。
2. 当社製品または本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。
3. 当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を組み込んだ製品の輸出入、製造、販売、利用、配布その他の行為を行うにあたり、第三者保有の技術の利用に関するライセンスが必要となる場合、当該ライセンス取得の判断および取得はお客様の責任において行ってください。
5. 当社製品を、全部または一部を問わず、改造、改変、複製、リバースエンジニアリング、その他、不適切に使用しないでください。かかる改造、改変、複製、リバースエンジニアリング等により生じた損害に関し、当社は、一切その責任を負いません。
6. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。

   標準水準：　コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット等

   高品質水準：輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、金融端末基幹システム、各種安全制御装置等

   当社製品は、データシート等により高信頼性、Harsh environment 向け製品と定義しているものを除き、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙機器と、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することは想定していません。たとえ、当社が想定していない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。
7. あらゆる半導体製品は、外部攻撃からの安全性を 100%保証されているわけではありません。当社ハードウェア／ソフトウェア製品にはセキュリティ対策が組み込まれているものもありますが、これによって、当社は、セキュリティ脆弱性または侵害（当社製品または当社製品が使用されているシステムに対する不正アクセス・不正使用を含みますが、これに限りません。）から生じる責任を負うものではありません。当社は、当社製品または当社製品が使用されたあらゆるシステムが、不正な改変、攻撃、ウイルス、干渉、ハッキング、データの破壊または窃盗その他の不正な侵入行為（「脆弱性問題」といいます。）によって影響を受けないことを保証しません。当社は、脆弱性問題に起因しまたはこれに関連して生じた損害について、一切責任を負いません。また、法令において認められる限りにおいて、本資料および当社ハードウェア／ソフトウェア製品について、商品性および特定目的との合致に関する保証ならびに第三者の権利を侵害しないことの保証を含め、明示または黙示のいかなる保証も行いません。
8. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は、データシート等において高信頼性、Harsh environment 向け製品と定義しているものを除き、耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
10. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。かかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。
12. お客様が当社製品を第三者に転売等される場合には、事前に当該第三者に対して、本ご注意書き記載の諸条件を通知する責任を負うものといたします。
13. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。
14. 本資料に記載されている内容または当社製品についてご不明な点がございましたら、当社の営業担当者までお問合せください。

注 1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社が直接的、間接的に支配する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

(Rev.5.0-1 2020.10)

本社所在地

〒135-0061 東京都江東区豊洲 3-2-24（豊洲フォレシア）

www.renesas.com

お問合せ窓口

弊社の製品や技術、ドキュメントの最新情報、最寄の営業お問合せ窓口に関する情報などは、弊社ウェブサイトをご覧ください。

www.renesas.com/contact/

商標について

ルネサスおよびルネサスロゴはルネサス エレクトロニクス株式会社の商標です。すべての商標および登録商標は、それぞれの所有者に帰属します。