UG136：Silicon Labs *Bluetooth* ® C アプリケーション開発者ガイド（SDK v2.x 用）

このドキュメントは、Silicon Labs Bluetooth スタックを使用して、Silicon Labs Wireless Gecko 製品向け C ベース・アプリケーションを開発する際に非常に役立つ資料です。このガイドでは、Bluetooth スタックのアーキテクチャ、アプリケーション開発フロー、MCU コアおよびペリフェラルの使用と制限、スタック構成オプション、およびスタック・リソースの使用について説明します。このバージョンは、Silicon Labs Bluetooth ソフトウェア開発キット（SDK）バージョン 2.13.x 以降に適用されます。

本書は、Wireless Gecko 向け Bluetooth アプリケーションの開発において、Bluetooth スタック API リファレンス、Gecko SDK API リファレンス、および Wireless Gecko リファレンス・マニュアルに記載されていなかった部分を補うことを目的としています。本書では、開発者が利用可能なハードウェア・リソースを最大限活用できるように、詳しく説明します。

要点

- プロジェクト構造と開発フロー
- Bluetooth スタックと Wireless Gecko の構成
- 割り込み処理
- イベントおよびスリープ管理
- リソースの使用方法と利用可能なリソース

目次

第 1 章 はじめに

本書は、Silicon Labs Bluetooth スタック用の C 開発者ガイドです。

本書は、開発についてさまざまな角度から説明するもので、Bluetooth スタックを使用する Wireless Gecko 製品向けに C 言語で開発を行う際の重要な資料になります。

本書では以下のトピックについて説明します。

- セクション 第 2 章 アプリケーション開発フロー では、アプリケーション開発フローについて説明します。
- セクション 第 3 章 プロジェクト構造 では、プロジェクト構造についてレビューします。
- セクション 第 4 章 Bluetooth スタックと Wireless Gecko デバイスの構成 では、プロジェクトに含まれるライブラリと、アプリケーション・コード内の実際の Wireless Gecko の構成について説明します。
- セクション 第 5 章 Bluetooth スタックのイベント処理 は、Silicon Labs Bluetooth スタックを使用する開発者にとって重要なセクションです。ここではイベント・ベースのアーキテクチャ内でアプリケーションがスタックとどのように同期して動作するかを説明します。
- セクション 第 6 章 割り込み とセクション 第 7 章 Wireless Gecko のリソース では、ペリフェラルとチップセット・リソースについて簡単に説明し、スタックの使用向けに予約されるリソース、割り込みの処理、スタックのメモリ・フットプリントとアプリケーションに使用可能なメモリについて取り上げます。

1.1 本バージョンについて

Silicon Labs の Bluetooth SDK の現在のバージョンは 2.13.x です。

現在サポートされているコンパイラおよび IDE のバージョンは以下のとおりです。

- **IDE:** Simplicity Studio 4、バージョン 4.1.11 以降。Bluetooth SDK v2.x は Simplicity Studio 5 とは互換性がありません。
- **コンパイラ:** IAR v8.30.1 および GCC 7.2.1

1.2 前提条件

本書では、Silicon Labs の Bluetooth SDK の最新バージョンが開発マシン (Windows、MAC OSX、または Linux) に適切にインストールされ、読者がクイック・スタート・ガイドおよび SDK の事例に精通していることを前提としています。また、読者は Bluetooth 技術の基礎を理解している必要があります。詳細については、『*UG103.14 : Bluetooth 技術の基礎*』を参照してください。

Silicon Labs Simplicity Studio 開発環境でアプリケーション例を使用して開始する手順については、『*QSG139 : Simplicity Studio を使用した Bluetooth 開発*』を参照してください。

第 2 章 アプリケーション開発フロー

次の図に、ファームウェア構造の概要を示します。開発者はアプリケーションをスタックの上に構築しますが、Silicon Labs はこのスタックをプリコンパイル済みのオブジェクト・ファイルとして提供し、これによってエンド・デバイスの Bluetooth 接続が可能になります。

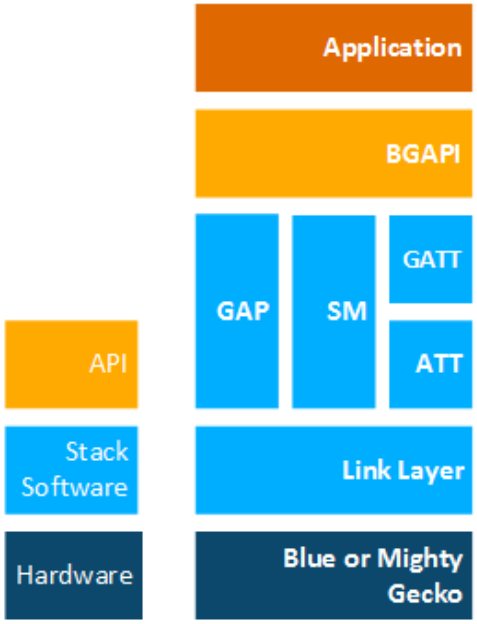


図 2.1. Bluetooth スタック・アーキテクチャのブロック図

Bluetooth スタックには以下のブロックが含まれています。

- **ブートローダ** – Gecko ブートローダはスタックの一部ではありませんが、Bluetooth SDK と共に提供されています。詳細については、『*UG266：Gecko ブートローダ・ユーザ・ガイド*』および『*AN1086：Silicon Labs Bluetooth アプリケーションでの Gecko ブートローダの使用*』を参照してください。ブートローディングの一般的な情報については、『*UG103.06：Bootloading Fundamentals（ブートローディングの基礎）*』を参照してください。
- **Bluetooth スタック** - リンク・レイヤ、汎用アクセス・プロファイル、セキュリティ・マネージャ、属性プロトコル、および汎用属性プロファイルで構成される Bluetooth の機能。
- **Bluetooth AppLoader** – ブートローダの後に起動するアプリケーション。ユーザ・アプリケーションが有効かどうかをチェックし、有効な場合は、AppLoader がアプリケーションを起動します。アプリケーション・イメージが無効な場合は、AppLoader が OTA プロセスを開始して有効なアプリケーション・イメージの受信を試みます。これを実行するには Gecko ブートローダが必要です。

2.1 アプリケーション構築フロー

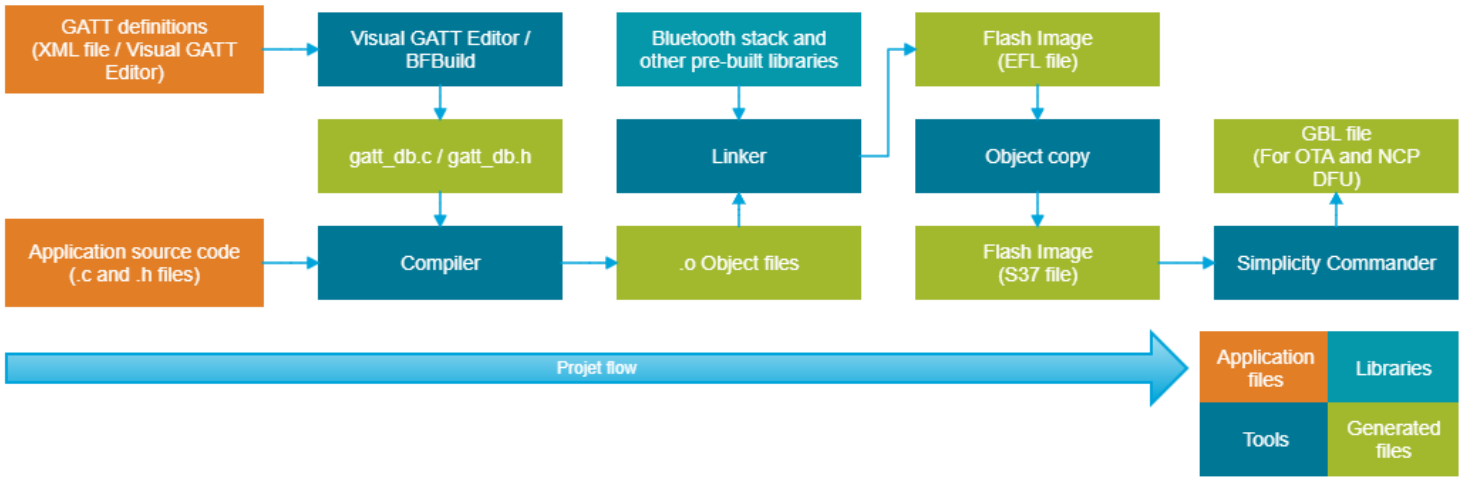


図 2.2. Bluetooth プロジェクトの構築フロー

プロジェクトを構築するには、まず、Bluetooth サービスと特性（GATT 定義）を定義し、Silicon Labs が提供する例、または空のプロジェクト・テンプレートからアプリケーション・ソース・コードを記述します。この手順については、『*QSG139：Simplicity Studio を使用した Bluetooth アプリケーション開発*』で説明しています。

SDK v2.1.0 以降では、Bluetooth サービスと特性を定義する 2 つの方法が用意されています。最初の方法は、Simplicity Studio の Visual GATT Editor GUI を使用します。これは、GATT を設計し、*gatt_db.c* および *gatt_db.h* を生成するためのグラフィカル・ツールです。また、このツールを使用して *.xml* ファイルと *.bgproj* GATT 定義ファイルをインポートできます。Visual GATT Editor は、Simplicity Studio プロジェクトで GATT を定義し生成するためのデフォルト・ツールです。

2 番目の方法は、*.xml* または *.bgproj* を『*UG118：Blue Gecko Bluetooth® Profile Toolkit 開発者ガイド*』に従って作成し、次にプリコンパイル・ステップで BGBuild 実行可能ファイルを使用して GATT 定義ファイルを .c および .h に変換する方法です。この方法は IAR Embedded Workbench プロジェクトで使用されます。

プロジェクトをコンパイルすると、オブジェクト・ファイルが生成されます。このファイルはその後、SDK で提供されるプリコンパイル・ライブラリとリンクされます。リンクにより、サポートされている Wireless Gecko デバイスに対してプログラム可能なフラッシュ・イメージが生成されます。

第 3 章 プロジェクト構造

このセクションでは、アプリケーション・プロジェクト構造と、プロジェクトに含める必要のある必須のリソースおよびオプションのリソースについて説明します。

3.1 Bluetooth ファイル

ライブラリ・ファイル

Bluetooth スタック・ライブラリは以下で構成されます。

- **binapploader.o**：Bluetooth AppLoader のバイナリ・イメージ。オプションの OTA（無線）機能を提供します。
- **binapploader_nvm3.o:** NVM3 に対応したシリーズ 1 用 Bluetooth AppLoader のバイナリ・イメージ。
- **libbluetooth.a**：Bluetooth スタックのライブラリ。
- **libmbedtls.a**：Bluetooth スタック用の mbed TLS 暗号化ライブラリ。
- **libpsstore.a:** Bluetooth スタックの PS ストア機能。EFR32BG2x デバイスでは使用できません。NVM3 を使用する必要があります。

RAIL

Bluetooth スタックは RAIL を使用して無線にアクセスします。RAIL ライブラリは Bluetooth スタックとリンクさせる必要があります。RAIL には、シングルプロトコル環境とマルチプロトコル環境に対応する、デバイス・ファミリごとの個別ライブラリがあります。RAIL ライブラリは Gecko SDK で提供されます。詳細については、『*UG103.13：RAIL Fundamentals （RAIL の基礎）*』およびその他の RAIL ドキュメントを参照してください。

Note: 無線モジュールを規制に準拠させるために、無線モジュール用の Bluetooth スタックを RAIL ライブラリおよびラジオ・モジュール用の構成ライブラリにリンクする必要があります。これらは、librail_module_<soc family><compiler>_release.a と librail_config<modulename>.a. です。

EMLIB および EMDRV

Bluetooth スタックは、EMLIB ライブラリと EMDRV ライブラリを使用して EFR32 ハードウェアにアクセスします。EMLIB および EMDRV のペリフェラル・ライブラリはソース・コードで提供され、このライブラリはプロジェクトに含める必要があります。EMLIB および EMDRV は Gecko SDK の一部です。EMLIB および EMDRV の詳細については、<Simplicity Studio Gecko SDK>\platform\bootloader\documentation\Gecko_Bootloader_API_Reference\index.html にある Gecko ブートローダ API のリファレンスと、<Simplicity Studio Gecko SDK>\platform\ のそれぞれのフォルダにあるドキュメントを参照してください。

スリープ・タイマ

スリープ・タイマは、ソフトウェア・タイマ、時間管理、および日付機能を提供するプラットフォーム・コンポーネントです。Bluetooth スタックはこれをディープ・スリープに使用します。スリープ・タイマはプロジェクトに含める必要があります。プラットフォーム・スリープ・タイマの資料を参照してください。

ヘッダ・ファイル

bg_version.h

このファイルには、Bluetooth スタックのバージョンが含まれます。

API ヘッダ・ファイル

これらのファイルは、Bluetooth スタック API を定義します。さまざまな使用状況に対応するファイルが 3 種類あります。これらのファイルのうち 1 つのファイルのみを含めてください。*native_gecko.h* は、ベアメタル Bluetooth アプリケーションで使用します。*ncp_gecko.h* は、NCP をサポートするために SoC アプリケーションを構築する際に使用します。*rtos_gecko.h* は、Micrium RTOS で使用します。

これらのファイルには 2 つの目的があります。最初の目的は、実際の Bluetooth スタック API と、そのスタックのコマンドとイベントを含めること、2 番目の目的は、構成、イベント、およびスリープ管理 API を Bluetooth スタックに提供することです。

構成、イベント、およびスリープ管理 API を以下で説明します。

errorcode_t gecko_init(const gecko_configuration_t*config)

この関数は、単一の引数（gecko_configuration_t 構造体へのポインタ）を取ります。この関数の目的は、構造体に提供されたパラメータで Bluetooth スタックを構成し初期化することです。構成オプションと gecko_init() の使用方法についてはセクション 4.2 gecko_stack_init() を使用した Bluetooth 構成で詳しく説明します。gecko_init() は、Bluetooth スタックを初期化するためにアプリケーションによって呼び出される必要があります。

この機能は便宜上備えられています。Bluetooth スタックのすべての機能を初期化します。構成の粒度をさらに高めるには、以下で説明するように gecko_stack_init() を使用します。

errorcode_t gecko_stack_init(const gecko_configuration_t*config)

この関数は、単一の引数（gecko_configuration_t 構造体へのポインタ）を取ります。この関数の目的は、構造体に提供されたパラメータで Bluetooth スタックを構成し初期化することです。関数 gecko_stack_init() が呼び出された後、各スタックが使用するコンポーネントを個別に初期化する必要があります。個別に行うことで、不要なスタック・コンポーネントを含めずにメモリを最適化することができます。

以下の API は、スタック・コンポーネントを個別に初期化するために使用できます。

- gecko_bgapi_class_dfu_init();
- gecko_bgapi_class_system_init();
- gecko_bgapi_class_le_gap_init();
- gecko_bgapi_class_le_connection_init();
- gecko_bgapi_class_gatt_init();
- gecko_bgapi_class_gatt_server_init();
- gecko_bgapi_class_hardware_init();
- gecko_bgapi_class_flash_init();
- gecko_bgapi_class_test_init();
- gecko_bgapi_class_sm_init();

struct gecko_cmd_packet*gecko_wait_event(void)

これは、イベントが Bluetooth スタックから送信されるのを待ち、イベントを受信するまでブロックする、ブロッキング関数です。イベントが受信されると、gecko_cmd_packet 構造体へのポインタが返されます。

Bluetooth スタック構成で EM スリープ・モードが有効になっている場合、Bluetooth スタックからイベントを受信しないと、デバイスは自動的に EM1 または EM2 モードになります。できるだけデバイスを消費電力の最も低いスリープ・モードにするには、gecko_wait_event() を使用するのが一番簡単な方法です。

Bluetooth スタックのイベント処理は、セクション 第 5 章 Bluetooth スタックのイベント処理 で詳しく説明します。

struct gecko_cmd_packet* gecko_peek_event(void)

これは、Bluetooth スタックから Bluetooth イベントを要求する、ノンブロッキング関数です。イベントが要求され、イベント・キューが空でない場合、gecko_cmd_packet 構造体へのポインタが返されます。イベント・キューにイベントがない場合は NULL が返されます。

このノンブロッキング・イベント・リスナを使用する場合、EM スリープ・モードは Bluetooth スタックで自動的に管理されないためアプリケーション・コードで管理する必要があります。このスリープ・モード管理は gecko_can_sleep_ms() 関数と gecko_sleep_for_ms() 関数で行いますが、これらは後で説明します。

スタックのイベント処理は、セクション 第 5 章 Bluetooth スタックのイベント処理 で詳しく説明します。

`int gecko_event_pending(void)`

この関数は、イベント・キューに保留中の Bluetooth スタック・イベントがあるかどうかを確認します。保留中の Bluetooth イベントが検出された場合、関数は 0 以外の値を返し、`gecko_peek_event()` または `gecko_wait_event()` によってイベントを処理する必要があることを示します。イベントが検出されない場合は、0 が返されます。

`uint32 gecko_can_sleep_ms(void)`

この関数を使用して、Bluetooth スタックをスリープ状態にできる時間を決定します。戻り値は、次の Bluetooth 動作が発生するまでスタックをスリープ状態にできるミリ秒数です。スリープ状態にできない場合は、0 が返されます。この関数は、ノンブロッキング `gecko_peek_event()` イベント処理でのみ使用されます。

`uint32 gecko_sleep_for_ms(uint32 max)`

この関数は、最大ミリ秒数（このミリ秒数は関数の単一パラメータで設定されます）スタックを EM スリープ状態にする場合に使用されます。戻り値は、実際にスリープ状態になったミリ秒数です。外部イベントによって、スタックのスリープ状態が解除される可能性があります。この関数は、ノンブロッキング `gecko_peek_event()` イベント処理でのみ使用されます。

native_gecko.h

このファイルは、RTOS を使用しないアプリケーションで使用します。直接関数呼び出しを使用して、Bluetooth スタックに IPC（プロセス間通信）を提供します。

ncp_gecko.h

このファイルは、ホストの NCP 機能を提供するアプリケーションで使用します。関数呼び出しとして NCP ヘッダを使用することによって Bluetooth スタックに IPC を提供します。

host_gecko.h および gecko_bglib.h

これらのファイルは、外部ホストのアプリケーションを開発する場合に使用します。`host_gecko.h` には API 定義が含まれ、`gecko_bglib.h` にはホスト・アプリケーションと BGAPI シリアル・プロトコル間のアダプテーション層が含まれます。

rtos_gecko.h

アプリケーションが Micrium OS 向けに構築されている場合は、`rtos_gecko.h` を使用します。Bluetooth スタックは、Micrium OS の独立したタスクであるため、Micrium OS の電源、スリープ、およびメモリ管理を使用します。`rtos_gecko.h` は、Micrium OS の任意のタスクから Bluetooth スタックを使用する場合の IPC 用のラッパーを提供します。このファイルには、Bluetooth スタック API、そのスタックのコマンドおよびイベント、Bluetooth スタックの構成 API が含まれています。

3.2 GATT データベース

GATT（汎用属性プロファイル）データベースは、Bluetooth デバイスの Bluetooth プロファイル、サービス、および特性を記述するための標準的な方法です。Silicon Labs Bluetooth スタックを使用する場合、GATT 定義は、Simplicity Studio の Visual GATT Editor GUI で直接編集するか、または XML で記述して、プレビルド・タスクとして BGBuild 実行可能ファイルに渡します。GATT データベースの作成方法と構文の詳細については、『*UG118：Blue Gecko Bluetooth® Smart Profile Toolkit 開発者ガイド*』を参照してください。

gatt_db.c および gatt_db.h

gatt_db.c は GATT データベースの構造とコンテンツを定義し、BGBuild.exe または Visual GATT Editor によって自動的に生成されます。gatt_db.h には、このデータベースと、ローカルな特性とサービスのハンドルが含まれます。GATT のタイプ定義は、gatt_db_def.h から gatt_db.h に自動的に含められます。

3.3 デバイス・ファームウェア・アップグレード

デバイス・ファームウェア・アップグレード (DFU) とは、シリアル・リンクまたは無線 (OTA) を介してアプリケーションをアップグレードするプロセスのことです。いずれの場合も、DFU のサポートを有効にするために、thr アプリケーションは以下のファイルを追加する必要があります。

application_properties.c

このファイルには、タイプ、バージョン、セキュリティなどのアプリケーション・イメージに関する情報をなど、アプリケーション・プロパティ構造体が含まれます。この構造体は Gecko ブートローダ API の application_properties.h で定義されます（<Simplicity Studio Gecko SDK>\platform\bootloader\documentation\Gecko_Bootloader_API_Reference\index.html の Gecko ブートローダ API リファレンスを参照）。事前に生成されたファイルは Simplicity Studio プロジェクトに含められ、アプリケーション固有のプロパティを含めるように変更できます。アプリケーション・プロパティには Gecko ブートローダ API を使用してアクセスできます。以下のメンバーは定義を変更して更新することができます。

```
// Version number for this application (uint32_t)
#define BG_APP_PROPERTIES_VERSION

// Capabilities of this application (uint32_t)
#define BG_APP_PROPERTIES_CAPABILITIES

// Unique ID (e.g. UUID or GUID) for the product this application is built for (uint8_t[16])
#define BG_APP_PROPERTIES_ID
```

Bluetooth AppLoader で OTA プロセスを使用する場合は、アプリケーション・プロパティ構造体をアプリケーション・ベクタ・テーブルの直後に配置する必要があります。これは、Bluetooth スタックから提供されるリンカ・ファイルを使用すると自動的に有効になります。

3.4 RTOS のサポート

Bluetooth スタックは Micrium RTOS でも実行できます。この場合、*native_gecko.h* は *rtos_gecko.h* と置き換えられ、プロジェクトに *rtos_bluetooth.c* ファイルと *rtos_bluetooth.h* ファイルが追加されます。

rtos_bluetooth.c と rtos_bluetooth.h

rtos_bluetooth.c および rtos_bluetooth.h は、Bluetooth スタックおよびその他の Micrium OS タスクとの IPC（プロセス間通信）の Micrium OS タスクを提供します。Micrium OS を使用する場合は、以下で説明する rtos_gecko.h ヘッダ・ファイルも含める必要があります。これは、任意の Micrium OS タスクから Bluetooth スタックを使用するための API IPC カプセル化を提供します。

gecko_configuration_t 構造体の Bluetooth スタック用に RTOS のサポートを構成する必要があります。config_flags フィールドには GECKO_CONFIG_FLAG_RTOS を設定しておく必要があります。これにより、Bluetooth スタックは、スリープ状態に直接移行するのではなく、Micrium OS によってスリープ状態に移行するようになります。scheduler_callback および stack_schedule_callback は、適切な関数を呼び出すように設定する必要があります。これらのコールバックは、対応するタスクをウェイク・アップする場合に使用します。

Micrium OS で使用する Bluetooth スタック構成は以下のとおりです。

```
.config_flags = GECKO_CONFIG_FLAG_RTOS, .scheduler_callback = BluetoothLLCallback, .stack_schedule_callback = BluetoothUpdate,
```

gecko_stack_init() を呼び出した後に、bluetooth_start_task() を呼び出すことができます。

```
void bluetooth_start_task(OS_PRIO ll_priority, OS_PRIO stack_priority);
```

これは、タスクの優先度をパラメータとして取得します。ll_priority はリンク・レイヤ、stack_priority は Bluetooth スタックです。システムの性能にはリンク・レイヤのタイムリーな実行が重要であるため、システムでリンク・レイヤの優先度を最も高くする必要があります。

3.5 マルチプロトコルのサポート

マルチプロトコル環境で Bluetooth スタックを使用する場合は、以下の関数を使用して Bluetooth スタックのマルチプロトコル機能を有効にする必要があります。

```
gecko_init_multiprotocol(const void *config);
```

現時点では、config パラメータは必ず NULL に設定され、今後の機能拡張に備えて予約されます。

マルチプロトコル環境で Bluetooth を使用するには、マルチプロトコルをサポートする RAIL ライブラリを使用する必要もあります。

3.6 ハードウェアのサポート

以下のファイルは Gecko SDK の一部であり、これらのファイルによって、ハードウェア固有機能のサポートが追加されます。

hal-config.h

このヘッダ・ファイルには、クロックや電源管理のための MCU ペリフェラルの初期設定のほか、UART、SPI などのペリフェラルの初期設定も含まれています。このファイルには、UART のボーレートのように、ペリフェラルのボードに固有でない設定のみが含まれ、UART の入出力ピンのようなボード固有の設定は含まれないことに注意してください。

init_mcu.c および init_mcu.h

これらのファイルには、クロックや電源管理のような MCU の内部設定を初期化する、デバイス初期化関数が含まれています。

init_board.c および init_board.h

これらのファイルには、ボードの外部部品を初期化する、ボード初期化関数が含まれています。GPIO の有効化、無線ボードの外部フラッシュの初期化などを行います。

init_app.c および init_app.h

これらのファイルには、アプリケーションに従って WSTK の外部部品を初期化する、アプリ初期化関数が含まれています。たとえば、WSTK の VCOM、センサー、および LCD ディスプレイを有効にします。

pti.c および pti.h

これらのファイルには、パケット・トレース・インターフェイスを有効にする、PTI 初期化関数が含まれています。

hal-config-board.h

このヘッダ・ファイルには、ボタンや LED ピン、UART および SPI ピンなどのボード初期化設定が含まれています。アプリケーションを Silicon Labs の無線ボード用に開発する場合、これらの設定は SDK で提供される例では正しく設定されていますが、カスタム・ハードウェア設計用のアプリケーションを作成する開発者は、必要に応じて設定を構成する必要があります。

bspconfig.h / bsphalconfig.h

BSP（ボード・サポート・パッケージ）ヘッダには、無線ボード固有の構成が含まれています。これらの構成は、WSTK の IO のトグル、スターター・キットの LCD ディスプレイの駆動など、WSTK 固有の関数のパラメータとして使用されます。Hardware Configurator ツールを使用する場合、例では `bsphalconfig.h` を使用します。それ以外の場合は、`bspconfig.h` を使用します。

mx25flash_spi.h

このヘッダ・ファイルには、一部の無線ボード（BRD4100A など）で SPI フラッシュ・チップを低消費電力モードに構成するための関数が含まれています。これは、たとえば、SPI フラッシュが低消費電力モードでない場合は、最小の EM2、EM3、または EM4 電流を達成できないため、スリープ電流を測定する際に役立ちます。

第 4 章 Bluetooth スタックと Wireless Gecko デバイスの構成

Wireless Gecko で Bluetooth スタックとアプリケーションを実行するには、MCU とそのペリフェラルが適切に構成されている必要があります。ハードウェアを初期化した後、gecko_init() 関数を使用してスタックも初期化する必要があります。

4.1 Wireless Gecko MCU とペリフェラルの構成

initMcu()

initMCU() 関数は MCU コアの初期化に使用します。この関数は発振器を起動し、デバイスのエネルギー・モードを構成します。ボードの設定に依存しないペリフェラルの初期化（タイマの初期化など）は、この関数に追加できます。この関数は、main() の最初に呼び出す必要があります。

initBoard()

initBoard() 関数は、外部フラッシュの初期化など、ボード機能の初期化に使用します。ボードの設計に依存するペリフェラルの初期化（GPIO の初期化や UART の初期化など）は、この関数に追加できます。この関数は、initMcu() の後に呼び出す必要があります。

initApp()

initApp() 関数は、WSTK で SPI ディスプレイを有効にするなど、アプリケーション固有の機能を初期化するために使用します。この関数は、initBoard() の後に呼び出す必要があります。

4.1.1 適応型周波数ホッピング

Bluetooth スタックが、適応型周波数ホッピング (AFH) を実装し、ETSI EN 300 328 規格に準拠するようになりました。AFH は、+10 dBm 以上の送信電力を使用するときに必要です。AFH によってチャンネルの輻輳が回避され、パフォーマンスが向上する場合もあります。

Bluetooth スタックで AFH を有効にするには、以下の初期化関数を呼び出す必要があります。

```
void gecko_init_afh();
```

マスタ/スレーブ接続では、両端で個々に AFH を使用できます。マスタが非適応型でも、スレーブは適応型でなければならないことがあります。この規格では、ブロックされたチャンネルで制御転送を使用できます。規格に準拠するために、ブロックされたチャンネルが使用中であることをスレーブが検知した場合は、スレーブはそのチャンネルで単一パケットのみを送信して、接続タイムアウトを防ぎます。

Note: 従来のアドバタイジングは、適応型周波数ホッピングを使用しません。従来のアドバタイジングは 3 つのチャンネルを使用し、AFH は ETSI 規格の要件を満たすために少なくとも 15 個のチャンネルを必要とします。アドバタイジングで AFH を有効にするには、拡張アドバタイジングを使用する必要があります。

4.1.2 Bluetooth クロック

クロック設定は initMcu_clocks() 関数で初期化されます。クロック設定には、チューニングなどのパラメータを含む発振器（HFXO、LFXO、および LFRCO）の初期化、クロック（HFCLK、LFCLK、LFA、LFB、LFE）の初期化、および発振器へのクロックの割り当てが含まれます。注：ペリフェラルのクロック（GPIO クロック、TIMER クロックなど）は、この関数では無効です。ペリフェラルを初期化する際に有効にする必要があります。

HFCLK

HFCLK は無線プロトコル・タイマ (PROTIMER) に使用されます。HFCLK は、精度が少なくとも ±50 ppm の範囲内に収まらなくてはならない高周波数クロックです。このクロックの精度を十分に確保するには外部水晶が必要です (HFXO)。

HFXO の初期化により、タイミングが重視される接続とスリープ管理を確保できるように外部水晶を構成します。HFXO は、高周波数クロック (HFCLK) として設定し、Wireless Gecko の HFXO 入力ピンに物理的に接続する必要があります。

LFCLK

LFCLK は低周波数クロックで、2 つの目的に使用されます。Bluetooth スタックでは、Bluetooth プロトコル・タイミング用に使用されます。また、スリープ・モードの間に時間をトラッキングするためにも必要です。

デバイスがスリープ・モードになる際、PROTIMER の現在の状態が保存されます。デバイスは、ウェイク・アップすると、経過したスリープ・クロックのティック数を計算し、それに応じて PROTIMER を調整します。無線ボードでは、PROTIMER が動作を続けていたように見えます。

このクロックの精度は、デバイスの動作モードによって異なります。アドバタイズやスキャンの場合、精度はあまり重要ではありませんが、接続がオープンである場合は、±500 ppm 以上の精度が必要です。このクロックは、精度要件に応じて、LFXO、PLFRCO (EFR32xG13 または xGM13)、あるいは LFRCO (EFR32xG2x または xGM2x) のいずれかで駆動できます。アプリケーションがアドバタイジングまたはスキャンのみを必要とする場合は、LFRCO をクロック・ソースとして使用することができます。ただし、Bluetooth 接続が必要な場合は、高精度モード (EFR32Xg22 または xGM22) で LFXO、PLFRCO (EFR32Xg13 または xGM13) あるいは LFRCO のいずれかを使用する必要があります。PLFRCO または LFRCO を使用する場合、クロックの精度は ±500 ppm に構成する必要があります。

クロックの精度は Bluetooth スタック構成構造で定義されます。4.2.5 Bluetooth スタック構成を参照してください。

デフォルトの構成では、LFXO が Wireless Gecko に接続され、LFCLK のクロック・ソースとして設定されます。高精度モードで PLFRCO または LFRCO のみが設計されている場合は、PLFRCO または LFRCO が接続され、クロック ソースとして設定されます。

高精度モードの LFXO、PLFRCO、または LFRCO のいずれも接続されていない場合、Bluetooth 接続は設計に必要ですが、スリープはアプリケーションから明示的に無効にする必要があります。セクション 4.2.4 スリープ で説明するように、LFCLK の精度が十分でない場合は、Bluetooth スタックを正確に動作させるために、スリープ・モードを無効にする必要があります。

HFRCODPLL

HFRCODPLL は、シリーズ 2 デバイスで Bluetooth スタックとともにシステム・クロックとして使用される高周波クロックです。EFR32xG21x では、HFRCODPLL を 80 MHz に構成し、システム・クロック・ソースとして設定する必要があります。

```
CMU_HFRCODPLLBandSet(cmuHFRCODPLLFreq_80M0Hz); CMU_ClockSelectSet(cmuClock_SYSCLK, cmuSelect_HFRCODPLL);
```

CTUNE

下の例では、HFXO と LFXO の両方の水晶チューニング (CTUNE) 設定が、Silicon Labs の Bluetooth モジュール、基準設計、および無線ボードのすべてで動作するようにデフォルトで設定されています。ただし、最終製品の設計では、デバイスごとに、または設計ごとに、個別に水晶校正が必要になる場合があります。CTUNE の値は、initMcu_clocks() 関数の hfxoInit.ctuneSteadyState と lfxoInit.ctune の設定を含む設計に応じて調整することができます。

```
// Initialize HFXO
CMU_HFXOInit_TypeDef hfxoInit = BSP_CLK_HFXO_INIT;
hfxoInit.ctuneStartup = BSP_CLK_HFXO_CTUNE;
hfxoInit.ctuneSteadyState = BSP_CLK_HFXO_CTUNE;
CMU_HFXOInit(&hfxoInit);
```

HFXO と LFXO の構成の詳細については、『EFR32 リファレンス・マニュアル』を参照してください。

デフォルトの HFXO CTUNE 値

システムは、次の論理的順序を使用して、デフォルトの HFXO CTUNE 値について複数のソースをチェックします。

1. CTUNE PSKEY が設定されています。このキーには ID 50（16 進数で 32）と、16 ビットの CTUNE 値の 2 バイトのデータが含まれています。これは、BGAPI コマンド cmd_flash_ps_save でプログラムすることができます。

2. 校正値が DEVINFO に存在します。一部のモジュールには、工場出荷時にプログラムされた値が DEVINFO-page に含まれています。

3. 製造トークンがユーザ・データ・ページに存在します。これは開発者によってプログラムされます。または、ボード EEPROM にこの値が含まれる場合は Simplicity Studio によって自動的に設定されます。このトークンは、2 バイトで構成されます。EFR32xG1x デバイスのユーザ・データ・ページ、または EFR32xG21 デバイスの最後のフラッシュ・ページの開始アドレスからのオフセット 0x0100 の位置にあります。フルフラッシュ・マッピングに関する特定の EFR バリアントについては、『EFR32 リファレンス・マニュアル』を参照してください。

4. プロジェクトの生成時に無線ボードが選択されている場合は、ボードのヘッダ・ファイルのデフォルト値を使用します。

5. 何も見つからない場合は、CMU ヘッダ・ファイルのデフォルト値を使用します。

注：Bluetooth スタックでは 38.4 MHz HFXO 周波数のみをサポートしています。その他の HFXO 周波数はサポートされていません。

4.1.3 DC-DC 構成

DC-DC を備えたデバイスの場合、構成は initMCU() 関数で設定します。SDK の例では、DC-DC 構成が Silicon Labs の Bluetooth モジュール、無線ボード、および基準設計で動作するように設定されていますが、カスタム設計では個別に DC-DC 設定を行う必要がある場合があります。これらのカスタム設定は *hal-config-board.h* で設定できます。

```
#define BSP_DCDC_INIT                                                  ¥
  {                                                                       ¥
    emuPowerConfig_DcdcToDvdd,    /* DCDC to DVDD */                      ¥
    emuDcdcMode_LowNoise,         /* Low-noise mode in EM0 */             ¥
    1800,                         /* Nominal output voltage for DVDD mode, 1.8V  */  ¥
    15,                           /* Nominal EM0/1 load current of less than 15mA */ ¥
    10,                           /* Nominal EM2/3/4 load current less than 10uA  */ ¥
    200,                          /* Maximum average current of 200mA
                                     (assume strong battery or other power source) */      ¥
    emuDcdcAnaPeripheralPower_DCDC,/* Select DCDC as analog power supply (lower power) */ ¥
    160,                          /* Maximum reverse current of 160mA */                   ¥
    emuDcdcLnCompCtrl_1uOF,       /* 1uF DCDC capacitor */                                 ¥
  }
```

DC-DC の構成の詳細については、『EFR32 リファレンス・マニュアル』の第 11 章と、『*AN0948：電力構成と DC-DC*』を参照してください。

4.1.4 LNA

低ノイズ・アンプ (LNA) は、信号対ノイズ比を大幅に低下させることなく超低電力信号を増幅する電子アンプです。LNA によって RF 感度が向上します。

一部の MGM12P モジュールでは、フロントエンド・モジュール（FEM）の一部として LNA がオンボードで提供されています。これらのモジュールで LNA を使用するには、FEM を正しく構成して有効にする必要があります。FEM は、プレフィックス BSP_FEM_ を使用して *hal-config-board.h* で構成します。

ボードが FEM をサポートしている場合、FEM は initBoard() 関数内の initFem() で初期化されます。

4.1.5 定期的なアドバタイジング

定期的なアドバタイジングにより、複数のリスナを単一のアドバタイジング・デバイスと同期させることができます。そのため、マルチキャストの形式になります。

各リスナは、データを受信する前にアドバタイジング・デバイスと同期する必要があります。定期的なアドバタイジングはリスニング・デバイス上のスキャナを使用して、アドバタイジング・デバイスとの同期を確立します。同期後に、スキャナは停止します。これにより、ブロードキャスト・アドバタイズ用のリスニングの場合、フルタイムでスキャナを使用するよりも電力効率が高まります。

定期的なアドバタイジングは、定期的なアドバタイザの役割と、リスニング側での定期的なアドバタイジングの同期という 2 つのコンポーネントで構成されます。これら 2 つのコンポーネントはそれぞれ独立しており、個別に初期化する必要があります。

定期的なアドバタイザ

Bluetooth 構成の `max_advertisers` では、定期的なアドバタイザの最大数も構成します。

Bluetooth スタックで定期的なアドバイザを有効にするには、ジェネリック `gecko_init` 関数の後で、以下の初期化関数を呼び出す必要があります。

```
void gecko_init_periodic_advertising();
```

定期的なアドバタイジングの同期

Bluetooth 構成の `max_periodic_sync` を使用して、Bluetooth スタックがサポートする必要がある同期の最大数を構成します。

Bluetooth スタックで定期的なアドバタイジングの同期を有効にするには、ジェネリック `gecko_init` 関数の後で、以下の初期化関数を呼び出す必要があります。

```
void gecko_bgapi_class_sync_init();
```

このコマンドは、BGAPI 同期クラスも初期化して、使用できるようにします。

4.1.6 PTI

PTI（パケット・トレース・インターフェイス）は Wireless Gecko SoC の組み込みブロックで、上下双方向の無線パケットを生データとしてデバッグ・インターフェイスにルーティングします。これらのパケットを Simplicity Studio のネットワーク・アナライザで取り込んで表示することができます。ネットワーク・アナライザには、Bluetooth パケット用のデコーダがあり、Bluetooth ネットワークのデバッグ、分析、および測定に使用できます。

PTI は、`initApp()` 関数内の `configEnablePti()` で初期化されます。ボーレートは HAL_PTI_BAUD_RATE 定義を使用して *hal-config.h* で設定でき、ピンは接頭辞 BSP_PTI_ を含む定義を使用して *hal-config-board.h* で構成できます。

Bluetooth 2.6.x 以降、PTI は RAIL によって提供される関数で構成されます。

4.1.7 送信電力

Bluetooth の送信電力は、無線、ソフトウェア構成、RF パス・ゲイン補償、および適応型周波数ホッピング (AFH) の使用状況により許容される最大電力で決まります。

ETSI EN 300 328 規格では、トランスミッタ電力が +10 dBm 以上の場合に AFH を使用する必要があります。

適応性要件に従って抑制されている場合、最大許容電力は +10 dBm 未満に制限されます。ETSI 規格では、AFH に少なくとも 15 個のチャンネルを使用することが求められます。この要件により、十分な数のチャンネルを使用できない場合、従来のアドバタイジング、スキャン応答、および接続に +10 dBm 以上を使用することはできません。

4.1.8 ホワイトリスト登録

ホワイトリスト登録は、デバイスのフィルタリングに使用します。現在は、デバイスの検出時にのみサポートされています。接続要求、アドバタイジング時のリモート・デバイスからのスキャン要求、および接続の開始は、ホワイトリストによって制限されません。

ホワイトリストのサイズは、結合されているデバイスの最大数の構成と一致します。ホワイトリスト登録の使用時に、結合されているデバイスの最大数が変更された場合は、新しい設定が有効になる前にデバイスをリセットする必要があります。

結合されているデバイスは、自動的にホワイトリストに追加されます。また、これらは BGAPI コマンド `gecko_cmd_sm_add_to_whitelist()` により手動で追加することができます。

ランダム・アドレスの解決はサポートされていません。解決可能なランダム・アドレスを使用するデバイスは、スキャン時に表示されません。ほとんどの Android および iOS 電話は解決可能なランダム・アドレスを使用しているため、これらのデバイスは、デバイスの検出時にホワイトリスト登録機能によって効率的にブロックされます。

Bluetooth スタックでホワイトリスト登録を有効にするには、ジェネリック `gecko_init` 関数の後で、以下の初期化関数を呼び出す必要があります。

```
void gecko_init_whitelisting();
```

この関数が有効になっている場合は、BGAPI コマンド `gecko_cmd_le_gap_enable_whitelisting()` によって実行時に有効/無効にすることができます。

4.1.9 Wi-Fi の共存

Wi-Fi の共存 (COEX) は、送信に無線を使用できるプロトコルを Bluetooth と Wi-Fi が判別するプロトコルです。これを有効にすると、Wi-Fi および Bluetooth の性能が向上します。COEX は、接頭辞 `BSP_COEX_` および `HAL_COEX_` が付く定義を使用して *hal-config-board.h* で構成します。

COEX を有効にするには、`gecko_stack_init()` の後に以下の関数を呼び出します。

```
gecko_initCoexHAL();
```

COEX は、Wi-Fi IC への GPIO インターフェイスを実装しています。これは、EMLIB `em_gpio.c` および EMDRV `gpiointerrupt.c` によって異なるため、両方のファイルをプロジェクトに含める必要があります。

4.2 gecko_stack_init() を使用した Bluetooth 構成

gecko_stack_init() 関数は、スリープ・モード構成、接続用のメモリ割り当て、OTA 構成などを含む、Bluetooth スタックを構成するために使用されます。Bluetooth スタックが構成されていない場合は、どの Bluetooth スタック関数も使用することはできません。

Bluetooth スタック構成の例：

```
uint8_t bluetooth_stack_heap[DEFAULT_BLUETOOTH_HEAP(MAX_CONNECTIONS)];
static const gecko_configuration_t config = {
  .config_flags=0,
  .sleep.flags=SLEEP_FLAGS_DEEP_SLEEP_ENABLE,
  .bluetooth.max_connections=MAX_CONNECTIONS,
  .bluetooth.heap=bluetooth_stack_heap,
  .bluetooth.heap_size=sizeof(bluetooth_stack_heap),
  .bluetooth.sleep_clock_accuracy = 100, // ppm
  .gattdb=&bg_gattdb_data,
  .ota.flags=0,
  .ota.device_name_len=3,
  .ota.device_name_ptr="OTA",
  .max_timers=4
};
```

gecko_stack_init() 関数の構成オプションは、スリープの有効化/無効化、Bluetooth 接続カウント、ヒープ・サイズ、スリープ・クロック精度、GATT データベース、OTA 構成、および PA 構成です。

関数 gecko_stack_init() が呼び出された後、使用される各スタック・コンポーネントを個別に初期化する必要があります。個別に行うことで、不要なスタック・コンポーネントを含めずにメモリを最適化することができます。

以下の API は、スタック・コンポーネントを個別に初期化するために使用できます。

gecko_bgapi_class_dfu_init()	デバイス・ファームウェア・アップグレード (dfu) API を有効にします。
gecko_bgapi_class_system_init()	ローカル・デバイス（システム）API を有効にします。
gecko_bgapi_class_le_gap_init()	汎用アクセス・プロファイル (gap) API を有効にします。
gecko_bgapi_class_le_connection_init()	接続 API を介して、接続の確立、パラメータ設定、および切断手順を管理できます。
gecko_bgapi_class_gatt_init()	gatt API を介して、リモート GATT サーバ内の属性を参照/管理できます。
gecko_bgapi_class_gatt_server_init()	gatt_server API を介して、ローカル GATT データベース内の属性を参照/管理できます。
gecko_bgapi_class_hardware_init()	ソフトウェア・タイマのアクセスおよび構成を有効にします。
gecko_bgapi_class_flash_init()	フラッシュ・メモリのユーザ・データを管理するために使用できる固定ストア・コマンド（フラッシュ）API を有効にします。
gecko_bgapi_class_test_init()	DTM テスト API を有効にします。
gecko_bgapi_class_sm_init()	セキュリティ・マネージャ (sm) API を有効にします。
gecko_bgapi_class_util_init()	*atoi*、*itoa* などのユーティリティ関数 API を有効にします。

4.2.1 CONFIG_FLAGS

現在のフラグ：

GECKO_CONFIG_FLAG_RTOS	1 = アプリケーションは RTOS を使用します。クロック、ベクタ、TEMPDRV、またはスリープは RTOS から提供されるためスタックでは構成しません。

4.2.2 Mbedtls

スタックによって使用される Mbedtls 暗号化ライブラリは、サポートされるアルゴリズムや、実装でハードウェア・アクセラレーションを使用するか、またはソフトウェアで実行するかどうかを定義する構成ファイルを使用して構成されます。Mbedtls 構成ファイル・パスは、#define MBEDTLS_CONFIG_FILE を使用して指定されます。デフォルトの構成ファイル mbedtls_config.h は SDK 内にあり、構成を変更する必要がある場合はテンプレートとして使用する必要があります。

4.2.3 マルチプロトコル優先度構成

マルチプロトコル環境で Bluetooth スタックが他のプロトコルと共に使用されている場合は、RAIL の Bluetooth 優先度設定を変更して、使用事例を最適化する必要がある場合があります。

アプリケーションが構成構造体を割り当てて、Bluetooth スタックに提供する必要があります。

```
gecko_bluetooth_ll_priorities custom_priorities; static const gecko_configuration_t config = { // .bluetooth.linklayer_priorities = &custom_priorities, // };
```

GECKO_BLUETOOTH_PRIORITIES_DEFAULT 定数により、gecko_bluetooth_ll_priorities 構造体をデフォルト状態に初期化する必要があります。

gecko_bluetooth_ll_priorities 構造には、以下のフィールドが含まれています。

- scan_min & scan_max - スキャン動作の優先度範囲。
- adv_min & adv_max - アドバタイズ動作の優先度範囲。
- conn_min & conn_max - 接続パケットの優先度範囲。
- init_min & init_max - 接続開始の優先度範囲。
- threshold_coex - 優先度信号を発生するしきい値レベル (COEX が有効な場合にのみ使用)。
- rail_mapping_offset - Bluetooth 優先度が配置されている、RAIL 優先度レベル。
- rail_mapping_range - Bluetooth 優先度が配置されている、RAIL 優先度範囲。

各優先度範囲で、0 が最大優先度、0xff が最小優先度です。Bluetooth 優先度は、RAIL 優先度とは異なります。つまり、Bluetooth では、0 と 0xff の間に独自のスペースがあり、そこにすべての Bluetooth 優先度が配置されます。Bluetooth 優先度を RAIL 優先度にマッピングするには、フィールド rail_mapping_offset と rail_mapping_range の値を使用して一次方程式を作成します。

```
RAIL_priority=(BT_priority/0xFF)*rail_mapping_range+rail_mapping_offset
```

4.2.4 スリープ

Wireless Gecko のスリープ・モード EM2（エネルギー・モード 2）は gecko_init() 関数で有効にする必要があります。スリープ・フラグは gecko_configuration_t 構造体の一部です。SLEEP_FLAGS_DEEP_SLEEP_ENABLED フラグを、スリープを有効にするように設定する必要があります。セクション 第 5 章 Bluetooth スタックのイベント処理 で説明するように、ブロッキング・イベントではスリープ・モードはスタックにより自動的に処理されます。

gecko_configuration_t 構造体 (main.c) でスリープを有効にする例：

```
.sleep.flags = SLEEP_FLAGS_DEEP_SLEEP_ENABLE    // EM sleeps enabled
```

スリープ・モードを使用するには、正確な 32 kHz 低周波数クロック (LFCLK) がハードウェアに必要です。Bluetooth スタックで正確なスリープ・クロックを使用できず、アプリケーションが Bluetooth 接続をサポートする必要がある場合は、低消費電力スリープ・モードにすることはできません。低消費電力スリープ·モードが必要ないアプリケーションの場合は、LFXO または LFRCO は外すことができますが、Gecko 構成構造体のスリープ・フラグは、次のように設定する必要があります。

```
  .sleep.flags = 0,                                   // Sleeps disabled
```

実行時のスリープを無効にする

実行時にスリープを無効にする必要がある場合は、スリープ・ドライバ関数 SLEEP_SleepBlockBegin(sleepEM2) を呼び出すことで無効にできます。EM2 ディープ・スリープ・モードを再度有効にするには、SLEEP_SleepBlockEnd(sleepEM2) を使用します。EM2 が無効になっている（ブロックされている）間は、スタックは EM0 と EM1 を切り替えます。詳細については、ナレッジ・ベース記事「Bluetooth スタックによるエネルギー・モードの使用」を参照してください。

4.2.5 Bluetooth スタック構成

スタック・メモリ

Bluetooth スタックは、内部メモリ管理を使用して、各接続および内部データ・バッファに対してメモリを割り当てます。このメモリは、アプリケーションが割り当てて Bluetooth スタックに渡す必要があります。メモリのサイズは接続数によって異なります。C マクロ DEFAULT_BLUETOOTH_HEAP() は、必要なメモリのデフォルト・サイズをバイト単位で算出します。

以下の例では、bluetooth_stack_heap アレイを割り当てて Bluetooth スタックに渡します。

```
uint8_t bluetooth_stack_heap[DEFAULT_BLUETOOTH_HEAP(MAX_CONNECTIONS)];
        static const gecko_configuration_t config = { // .bluetooth.heap =
        bluetooth_stack_heap, .bluetooth.heap_size = sizeof(bluetooth_stack_heap), // };
```

接続数

Bluetooth 同時接続の絶対最大数は 8 です。接続管理に割り当てられているメモリの容量によって、接続数はさらに制限されます。メモリは gecko_init() での初期化中に割り当てられます。C 定義 MAX_CONNECTIONS を定義すると接続数を設定できます。前述のように、同一の定義で Bluetooth スタックのメモリ・サイズも計算できます。その場合、MAX_CONNECTIONS は、さらに、構成構造体の .bluetooth.max_connections フィールドの Bluetooth スタックに渡されます。

Bluetooth 接続を 1 つに制限する例。

```
#define MAX_CONNECTIONS 1
```

接続の RAM 使用量の詳細については、7.3.2 Bluetooth 接続プールを参照してください。

スリープ・クロックの精度

Bluetooth スタックは、.sleep_clock_accuracy を使用してスリープからのウェイクアップ時間を最適化します。単位は ppm（100 万分の 1）です。この値が大きすぎると、Bluetooth スタックがスリープからウェイクアップするタイミングが早すぎて実際のイベントを待つことになり、余計な電力を消費する原因となります。この値が小さすぎると、Bluetooth スタックがウェイクアップするのが遅すぎて接続イベントを逃し、接続が中断する原因となります。

この値を定義していない場合または 0 に設定した場合は、デフォルト値の 250 ppm が使用されます。

以下に、スリープ・クロックの精度を設定する例を示します。

```
.bluetooth.sleep_clock_accuracy = 100, // ppm
```

アドバタイザ

アドバタイズ・セットの最大数は、この構成オプションによって定義できます。これらのセットを使用して複数のアドバタイザを開始できます。この構成オプションでは、定期的なアドバタイズの最大数も構成します。アドバタイズのコンテキストごとに約 60 バイトの RAM が割り当てられます。

```
.bluetooth.max_advertisers = 5;  //!< Maximum number of advertisers to support, if 0 defaults to 1
```

Note: 接続可能なアドバタイズの最大数は MAX_CONNECTIONS によって制限されます。

同期アドバタイズ

サポートされる同期アドバタイズの最大数を構成する必要があります。コンテキストごとに約 40 バイトの RAM が割り当てられます。

```
.bluetooth.max_periodic_sync = 5; //! <Maximum number of synhronous advertisers to support. Default is 0, none supported
```

4.2.6 OTA 構成

ファームウェア・アップグレードの一部は Bluetooth AppLoader アプリケーションにより処理されるため、Bluetooth 無線 (OTA) ファームウェアのアップグレードに対応しています。

OTA モードでは、.ota.flags 構成フィールドを使用します。現時点で用意されているオプションは、GECKO_OTA_FLAGS_RANDOM_ADDRESS だけです。このオプションでは、パブリック・アドレスではなく、スタティック・ランダム・アドレスを使用するために OTA を設定します。

Wireless Gecko が AppLoader の OTA モードになっていると、そのデバイス名とデバイス名の長さを gecko 構成構造体を使用して構成することができます。

```
  .ota.device_name_len = 3,     // OTA name length
  .ota.device_name_ptr = "OTA",    // OTA Device Name
```

最後に、デバイスを OTA DFU モードに確実に設定して、信頼できるデバイスのみに機能を制限する必要があります。

OTA ファームウェア・アップデートの詳細については、『*UG266 : Silicon Labs Gecko ブートローダ・ユーザ・ガイド*』および『*AN1086 : Silicon Labs Bluetooth アプリケーションでの Gecko ブートローダの使用*』を参照してください。

4.2.7 PA

EFR32 SoC ベースの設計では、PAVDD（パワー・アンプ電圧レギュレータ VDD 入力）を DC/DC の出力から供給するか、または 3.3 V 電源から直接供給できます。

Bluetooth スタック構成は、デフォルトで DC/DC を PAVDD 入力として使用します。PAVDD を 3.3 V 電源から供給する場合は、.pa.input フィールドを定義する必要があります。

Bluetooth スタックは自動的に高電力 PA を選択します（使用可能な場合）。pa_mode 構成を 1 に設定すると、Bluetooth スタックで常に低電力 PA が使用されるように構成されます。

```
.pa.config_enable = 1, // PA Configuration is enabled .pa.input = GECKO_RADIO_PA_INPUT_VBAT, // PAVDD is upplied from an 3.3 V power supply .pa.pa_mode=0 // selects high power PA if available
```

4.2.8 ソフトウェア・タイマ

使用可能なソフトウェア・タイマの最大数を構成できます。各タイマは、実装するスタックからのリソースを必要とします。使用事例によっては、ソフト・タイマの数が増えると性能が低下する場合があります。

```
.max_timers = 4; // Maximum number of soft timers, up to 16, Default: 4
```

4.2.9 RF 経路

ゲイン

アプリケーションは、RX と TX の RF パス・ゲイン値を個別に定義することができます。

Bluetooth スタックは、トランスミッタ電力を調整するときに TX RF パス・ゲインを考慮します。これにより、アンテナから放射される電力がアプリケーション要求と一致します。たとえば、アプリケーションによって要求される最大電力が +10 dBm であり、パス・ロスが -1 dBm である場合、ピンでの実際の電力は +11 dBm になります。

Bluetooth スタックからの RSSI レポートを補償するために、RX RF パス・ゲインが使用されます。

```
.rf.tx_gain = -20; // RF TX path gain in unit of 0.1 dBm .rf.rx_gain = -18; // RF RX path gain in unit of 0.1 dBm
```

出力の選択

EFR32XG21 SoC ベース設計では、RF 出力を選択できます。

```
.rf.flags = GECKO_RF_CONFIG_ANTENNA; // enabling output configuration .rf.antenna = 0; // desired output,
```

正しい値については、RAIL ヘッダ・ファイル rail_chip_specific.h のアンテナ経路の選択を参照してください。

第 5 章 Bluetooth スタックのイベント処理

Wireless Gecko 用の Bluetooth スタックはイベント駆動型アーキテクチャで、イベントはメイン while ループで処理されます。

5.1 ブロッキング・イベント・リスナ

gecko_wait_event() は、ブロッキング待機関数の実装であり、イベントがイベント・キューに入るまで待って、イベントをイベント・ハンドラに返します。この動作モードでは、デバイスと接続の同期を維持しながら、スリープを最も効率よく自動的に管理できるため、Bluetooth スタックではこのモードが推奨されます。

- gecko_wait_event() 関数は、イベントが受信されるまで、内部メッセージ・キューを処理します。
- 処理する保留中のイベントやメッセージがない場合、デバイスは EM1 または EM2 スリープ・モードになります。
- この関数は、受信したイベントを保持しながら、gecko_cmd_packet 構造へのポインタを返します。

以下のコード・スニペットは、iBeacon の例の簡単なメインの while ループを示しています。ここでは gecko_wait_event() を使用して、起動後にアドバタイズを設定します。

```
/* Main loop */
while (1) {
    struct gecko_cmd_packet* evt;

    /* Wait (blocking) for a Bluetooth stack event. */
    evt = gecko_wait_event();

    /* Run application and event handler. */
    switch (BGLIB_MSG_ID(evt->header))
    {
    /*  This boot event is generated when the system is turned on or reset. */
    case gecko_evt_system_boot_id:

      /* Initialize iBeacon ADV data */
      bcnSetupAdvBeaconing();
      break;

    /* Ignore other events  */
    default:
      break;
    }
```

5.2 ノンブロッキング・イベント・リスナ

この動作モードでは、より多くの手動調整が必要です。たとえば、スリープ管理はアプリケーションで行う必要があります。一部の使用事例では、ノンブロッキング動作が必要になります。

- gecko_peek_event() 関数は、イベントが受信されるまで、またはすべてのメッセージが処理されるまで、内部メッセージ・キューを処理します。
- この関数は、受信したイベントを保持しながら gecko_cmd_packet 構造へのポインタを返します。またはキューにイベントがない場合は NULL を返します。

5.2.1 スリープとノンブロッキング・イベント・リスナ

アプリケーションがノンブロッキング gecko_peek_event() 関数を使用してイベント・ハンドラを作成する場合、スリープ実装も異なります。アプリケーションは、gecko_can_sleep_ms() を使用してデバイスをスリープ状態にできる時間についてスタックに対してクエリを実行し、次に gecko_sleep_for_ms() 関数を使用してその時間スリープ状態になるように設定する必要があります。gecko_can_sleep_ms() 関数または gecko_sleep_for_ms() 関数を呼び出す前に割り込みを無効にし、関数が実行された後に割り込みを有効にする必要があります。

Note: この重要なセクションには、他の機能を何も追加しないことが推奨されます。機能を追加すると、割り込みに遅延が生じ、性能が低下します。

以下の例は、ノンブロッキング・イベント処理を使用する場合のスリープ管理の実装方法を示しています。

```
/* Main loop */
while (1) {
    struct gecko_cmd_packet* evt;
    CORE_DECLARE_IRQ_STATE;

    /* Poll (non-blocking) for a Bluetooth stack event. */
    evt = gecko_peek_event();

    /* Run application and event handler. */
    if(evt != NULL){
      switch (BGLIB_MSG_ID(evt->header))
      {
      /* This boot event is generated when the system is turned on or reset. */
      case gecko_evt_system_boot_id:

        /* Initialize iBeacon ADV data */
        bcnSetupAdvBeaconing();
        break;

      /* Ignore other events */
      default:
        break;
      }
    }
    CORE_ENTER_ATOMIC();                    // Disable interrupts

      /* Check how long the stack can sleep */
      uint32_t durationMs = gecko_can_sleep_ms();
      /* Go to sleep. Sleeping will be avoided if there isn't enough time to sleep */
      gecko_sleep_for_ms(durationMs);

    CORE_EXIT_ATOMIC();                     // Enable interrupts

}
```

5.2.2 イベント・リスナのアップデートに関する通知

アプリケーションの別のイベント・ループ内で Bluetooth イベント・ループを実行する必要が生じる場合があります。Bluetooth スタックにはコールバック・メカニズムがあり、Bluetooth スタックのイベント・リスナのアップデート要求についてアプリケーションに通知します。これを有効にするには、Bluetooth 構成の構造でコールバック関数を定義します。

Note: この stack_schedule_callback は割り込みコンテキストから呼び出されます。このコンテキストから gecko_peek_event または gecko_wait_event を呼び出さないようにしてください。アプリケーションのメイン・ループによる Bluetooth スタックのアップデートを有効にするには、アプリケーションでフラグを設定するか、別のメカニズムを使用する必要があります。

```
static const gecko_configuration_t config = { // .stack_schedule_callback = bluetooth_update // }; void bluetooth_update() { //set notification for application }
```

5.3 Micrium OS を使用したイベント・リスナ

Micrium OS でイベントを受信する場合、アプリケーションは別の手順を使用します。アプリケーションは、関数を呼び出してイベントを受信する代わりに、Micrium OS フラグを待機する必要があります。イベントは 1 つのタスクからのみ受信できます。

bluetooth_event_flags の Micrium OS フラグは、別のタスクに Bluetooth スタックの状態を通知する場合に使用します。アプリケーションは、BLUETOOTH_EVENT_FLAG_EVT_WAITING と BLUETOOTH_EVENT_FLAG_EVT_HANDLED のみを使用します。

アプリケーションのイベント・ハンドラは、BLUETOOTH_EVENT_FLAG_EVT_WAITING を待機する必要があります。

```
OSFlagPend(&bluetooth_event_flags, (OS_FLAGS)BLUETOOTH_EVENT_FLAG_EVT_WAITING 0,OS_OPT_PEND_BLOCKING + OS_OPT_PEND_FLAG_CONSUME, NULL,&os_err);
```

受信したイベントは bluetooth_evt で使用できます。

```
switch (BGLIB_MSG_ID(bluetooth_evt->header)) { ... }
```

イベントを処理した後は、次のイベントを処理できるように解放する必要があります。これを行うには、フラグ BLUETOOTH_EVENT_FLAG_EVT_HANDLED を立てて Bluetooth タスクに通知します。

```
OSFlagPost(&bluetooth_event_flags, (OS_FLAGS)BLUETOOTH_EVENT_FLAG_EVT_HANDLED, OS_OPT_POST_FLAG_SET, &os_err);
```

Note: アプリケーションが保留状態の場合、スリープおよび電源管理は、アプリケーションではなく Micrium OS によって自動的に処理されます。スタックを OSIdleConextHook() のスリープ･モードに設定するには、アプリケーションでループを使用する必要があります。この関数は、実行できる状態にあるタスクがない場合に、カーネルによって呼び出されます。

```
void SleepAndSyncProtimer(); void OSIdleContextHook(void) { while (1) { /* Put MCU in the lowest sleep mode available, usually EM2 */ SleepAndSyncProtimer(); } }
```

5.3.1 複数のタスクからのコマンド

Bluetooth コマンドは複数の Micrium OS タスクから送信できます。これを行うには、各タスクがコマンドの送信前に排他状態に移行し、送信後に排他状態を解除する必要があります。

Bluetooth スタックには、便利な 2 つの関数が用意されています。BluetoothPend は Micrium OS ミューテックスを取得し、BluetoothPost はミューテックスを解放します。

以下のコード・ブロックは、Bluetooth コマンドの前に Bluetooth のミューテックスを取得し、その後、そのミューテックスを解放します。

```
BluetoothPend(&err); //acquire mutex for Bluetooth stack gecko_cmd_gatt_server_send_characteristic_notification(0xff, gattdb_temp_measurement, 5, temp_buffer); BluetoothPost(&err);//release mutex
```

第 6 章 割り込み

無線割り込みまたは IO ピンからの割り込みによって、それぞれの割り込みハンドラでイベントが生成されます。イベントは、メッセージ・キューからのメイン・イベント・ループで後で処理されます。アプリケーションは、常に割り込みハンドラ内での処理時間を最小限にし、イベント・コールバックやメイン・ループで処理を行う必要があります。

一般に、割り込みスキームはイベント・ベースのプログラミング・アーキテクチャに従いますが、Bluetooth スタックには、他にはない重要な例外がいくつかあります。

- BGAPI コマンドは、割り込みコンテキストから呼び出せません。
- 割り込みコンテキストから呼び出せるのは、gecko_external_signal() 関数のみです。
- 前のコード例に示したように、gecko_sleep_for_ms(...) を呼び出す前に、割り込みを無効にする必要があります。

6.1 外部イベント

外部イベントは、すべてのペリフェラル割り込みを外部信号として取り込み、メイン・イベント・ループに渡してループ内で処理するために使用されます。外部イベント割り込みは、IO、コンパレータ、ADC など、さまざまなペリフェラル割り込みソースから発生します。どのような外部割り込みが発生したかをイベント・ハンドラに通知するために、信号ビット配列が使用されます。

- 外部信号の主な目的は、割り込みコンテキストからメイン・イベント・ループに対してイベントをトリガすることです。
- BGAPI イベント system_external_signal は、void gecko_external_signal(uint32 signals) 関数を呼び出すことにより生成できます。
- 関数 gecko_external_signal は割り込みコンテキストから呼び出せます。
- gecko_external_signal 関数の signals パラメータは system_external_signal イベントに渡されます。

```
/**
* Main
*/
void main()
{
  ...

      //Event loop
  while(1)
  {
    ...

      //External signal indication (comes from the interrupt handler)
    case gecko_evt_system_external_signal_id:
      // Handle GPIO IRQ and do something
      // External signal command's parameter can be accessed using
     // event->data.evt_system_external_signal.extsignals
      break;
    ...
  }
}

/**
* Handle GPIO interrupts and trigger system_external_signal event
*/
void GPIO_ODD_IRQHandler()
{
  static bool radioHalted = false;

  uint32_t flags = GPIO_IntGet();
  GPIO_IntClear(flags);

  //Send gecko_evt_system_external_signal_id event to the main loop
  gecko_external_signal(...);
  }
}
```

6.2 優先度

無線割り込みの優先度を最も高くすることを強くお勧めします。これはデフォルトの構成で、その他の割り込みはこれより低い優先度で処理されます。無線のデフォルトの割り込み優先度は 1、リンク・レイヤの場合の優先度は 2、USART の割り込み優先度は 3、その他の割り込みのデフォルトの優先度は 4 です。

アプリケーションで割り込みを無効にする必要がある場合は、PRIMASK レジスタではなく BASEPRI レジスタを使用することをお勧めします。BASEPRI レジスタは特定の優先度の割り込みを無効にしますが、PRIMASK はすべての割り込みを無効にします。BASEPRI レジスタを使用するように EMLIB Core を設定し、CORE_ENTER_ATOMIC() マクロと CORE_EXIT_ATOMIC() マクロで使用できます。

RTOS を使用しない場合、リンク・レイヤはアプリケーション・ソフトウェア全体での優先度を実現するために PendSV を使用します。RTOS を使用する場合は、リンク・レイヤが PendSV を使用することはありませんが、リンク・レイヤ・タスクはアプリケーション・タスクに対してより高い優先度を持つようになります。さらに、RTOS スケジューラがリンク・レイヤ・タスクに対して、アプリケーション・タスクを上回る優先度を与えます。

次の表では、Bluetooth スタック内の動作コンテキストが異なる 3 つのコンポーネントと、各コンポーネントの接続を確保するために割り込みを無効にする最大時間を説明します。

コンポーネント	説明	タイミング精度	動作コンテキス	最大 IRQ を無効化	タイミング要件が無視される場合
無線	タイム・クリティカルな低レベルの TX/RX 無線制御	マイクロ秒	無線 IRQ	< ~10 µs	パケットが送信または受信されていません。これにより、最終的に監視のタイムアウトおよび Bluetooth リンク・ロスが発生します。
リンク層	タイム・クリティカルな接続管理手順と暗号化	ミリ秒	PendSV IRQ*	< ~20 ミリ秒	リンク制御手順が時間内に処理されない場合、Bluetooth リンク・ログが発生することがあります。スレーブ側のチャンネル・マップ更新および接続更新タイミングはマスターによって制御されます。
ホスト・スタック	Bluetooth ホスト・スタック、セキュリティ・マネージャ、GATT	秒	アプリケーション	< 30 秒	SMP および GATT のタイムアウトは 30 秒に設定されています。そのタイムアウト内に動作が処理されない場合、Bluetooth リンク・ログが発生します。

*PendSV 割り込みは、RTOS を使用しない場合にのみ使用されます

第 7 章 Wireless Gecko のリソース

Bluetooth スタックで使用する Wireless Gecko のリソースの一部は、アプリケーションには使用できません。以下の表に、これらのリソースを示し、スタックがこれらをどのように使用するかを説明します。最初の 4 つのリソース（赤で表示）は、常に Bluetooth スタックで使用されます。

カテゴリ	リソース	ソフトウェアで使用	注
PRS	**PRS7**	PROTIMER RTC 同期	PRS7 は常に Bluetooth スタックで使用されます。
タイマ	**RTCC**	EM2 タイミング	スリープ・タイマは、デフォルトの構成で RTCC を使用します。 EFR32BG13 および EFR32BG2x では、スリープ・タイマが別のリソースを使用するように構成されていればアプリケーションが RTCC を使用できます。プラットフォーム・スリープ・タイマの資料 を参照
	PROTIMER	Bluetooth	アプリケーションは PROTIMER にアクセスできません。
無線	**RADIO**	Bluetooth	常に使用され、すべてのラジオ・レジスタが Bluetooth スタック用に確保されています。
GPIO	NCP	ホスト通信。	使用する機能（UART、RTS/CTS、ウェイク・アップ、およびホスト・ウェイクアップ）に応じて、2 ～ 6 本の I/O ピンを NCP の使用に割り当てることができます。 オプションで使用でき、NCP のユースケースに対してのみ有効です。
	PTI	パケット・トレース	2 ～ N 本の I/O ピン。 オプションで使用可能。
	TX イネーブル	TX アクティビティ表示	1 本の I/O ピン。 オプションで使用可能。
	RX イネーブル	RX アクティビティ表示	1 本の I/O ピン。 オプションで使用可能。
	COEX	Wi-Fi の共存	4 本の I/O ピン。 オプションで使用可能。
CRC	GPCRC	PS ストア	アプリケーションで使用できますが、使用する前に必ず GPCRC を再構成し、GPCRC クロックを CMU で無効にすることはできません。
フラッシュ	MSC	PS ストア	アプリケーションで使用できますが、MSC を無効にすることはできません。
CRYPTO	CRYPTO	Bluetooth リンクの暗号化	CRYPTO ペリフェラルは、mbedTLS 暗号ライブラリを介してのみアクセスできます。その他の方法ではアクセスできません。ライブラリはスタックとアプリケーション・アクセス間のスケジューリングを実行できます。

7.1 フラッシュ

アプリケーションおよび Bluetooth スタックはフラッシュ・メモリから実行されます。フラッシュは、次の図に示すように、ブートローダ用、Bluetooth AppLoader 用、アプリケーション + Bluetooth スタック用、および不揮発性メモリ用のブロックに分割できます。

- ブートローダは、Bluetooth スタックとアプリケーションのアップグレード性を有効にするために必要です。ブートローダは将来に対応した設計となっており、改良を行い機能を追加できます。独立したブートローダ・フラッシュを備えたデバイスの場合、ブートローダはその場所にあります。
- Bluetooth AppLoader は、アプリケーションの OTA の拡張性を提供します。これはオプション機能ですが、使用するにはブートローダを使用する必要があります。
- PS ストアおよび NVM3 は不揮発性のデータ・ストア（NVM）で、ここには Bluetooth スタックとアプリケーションの両方が Bluetooth 結合キー、アプリケーションの構成データ、ハードウェア構成などの恒久的データを保存できます。これらを同時に使用することはできません。PS ストアはシリーズ 1 デバイスのみでサポートされます。
- アプリケーションは、Bluetooth AppLoader と NVM の間にあります。Bluetooth スタックはアプリケーションとリンクしているライブラリです。Bluetooth スタックには、リンク・レイヤ、GAP、SM、ATT、および GATT レイヤなど、実際の Bluetooth ファームウェアが含まれます。
- ユーザ・データ・ページは、製造トークンの保存に使用されます。EFR32BG2X デバイスでは、メイン・フラッシュの最後にあります。

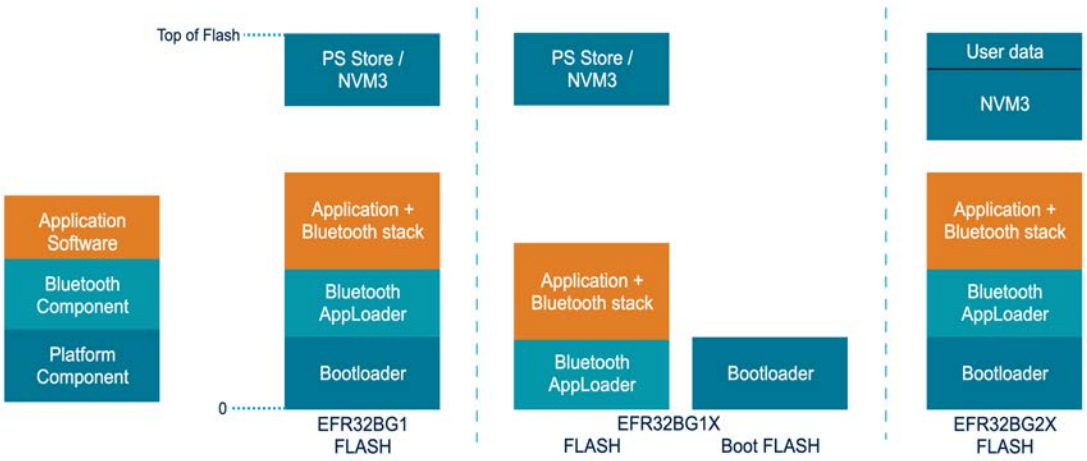


図 7.1. 独立したブートローダ・フラッシュがある場合とない場合のフラッシュの使用状況

次の表に、各ブロックのフラッシュの使用状況を示します。推定値は、使用事例、構成、アプリケーションのリソース、または SDK バージョンにより変わる可能性があります。

	コンパイラ	EFR32BG1	EFR32BG12	EFR32BG13	EFR32BG21	EFR32BG22
ブートローダ		16	16	16	16	24
Bluetooth AppLoader		40	44	46	48	56
soc-empty*	GCC	131	140	144	141	152
	IAR	131	140	143	141	153
soc-thermometer*	GCC	133	142	146	141	154
	IAR	133	142	145	142	155
PS ストア		4	4	4		
NVM3[+]		6	6	6	24	24
ユーザ・データ					8	8

**soc-empty* および *soc-thermometer* は、Bluetooth SDK で提供されているサンプル・アプリケーションです。これらは、高サイズの最適化によってコンパイルされます。GCC は `-Os` フラグを、 IAR は `-Ohz` フラグを使用します。

[+]NVM3 は、PS ストアの代わりに使用できます。これらを同時に使用することはできません。NVM3 では、少なくとも 3 つのフラッシュ・ページが必要です。これは、SDK の Bluetooth サンプル・アプリケーションのデフォルト構成です。詳細については、『*AN1135: 第 3 世代不揮発性メモリ（NVM3）データ・ストレージの使用*』を参照してください。

7.1.1 フラッシュ使用率の最適化

デッドコード削除

Bluetooth スタック・ライブラリは、リンカのデッドコード削除の最適化から利益を得られるように設計されています。この最適化によって、すべての不要なコードがアプリケーションから削除されます。

この最適化機能を最大限に活用するには、アプリケーションに不要な関数を呼び出さないようにしてください。これらには、Bluetooth スタックのすべての初期化関数が含まれます。

Bluetooth スタック・コンポーネントの選択的な初期化

`gecko_init()` 関数は、各スタック・コンポーネントを自動的に初期化します。より選択的な初期化を行うには、`gecko_stack_init()` を使用する必要があります。こうすることで、それぞれ必要なスタック・コンポーネントが個別に初期化されます。詳細については、「4.2 gecko_stack_init() を使用した Bluetooth 構成」を参照してください。

7.2 リンク方法

Bluetooth スタックは、ライブラリ・ファイルのセットとして届けられます。このアプリケーションは、Bluetooth スタックのライブラリを残りのアプリケーションとリンクさせます。それから、リンカは、フラッシュにロードできるアプリケーション・コードとデータを含んでいる ELF ファイルを作成します。

OTA DFU ファイルを生成する場合は、アプリケーションのコードとデータを ELF ファイルにある OTA DFU ファイル独自のセクションとリンクさせる必要があります。これは、Bluetooth スタックに提供されているリンカ・ファイルで自動的に行われます。

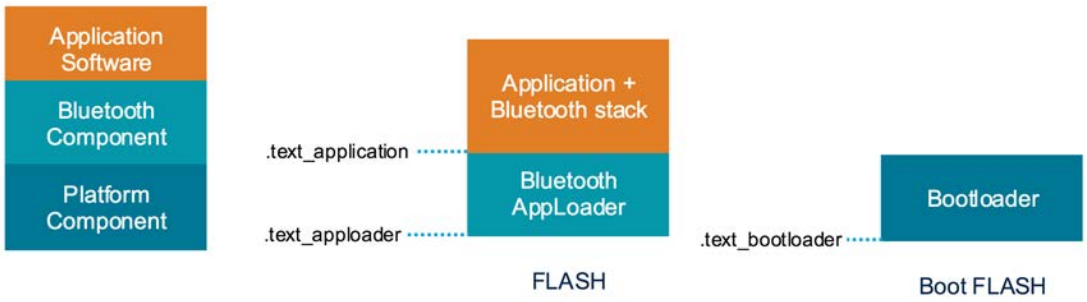


図 7.2. Sections Defined in the Linker File and Their Placement

リンカ・ファイルは、メイン・フラッシュとブートローダ用に、2 つのメモリ領域を定義します。個別のブートローダ・フラッシュが存在しない場合は、このリンカ・ファイルによってメイン・フラッシュの一部のメモリがブートローダ用に予約されます。Bluetooth AppLoader はメイン・フラッシュの先頭に配置され、すべてのライブラリを保有するアプリケーションは次の空きフラッシュ・ページから開始します。

OTA アップデートと、アップデートを有効にする方法の詳細については、『*UG266：Silicon Labs Gecko ブートローダ・ユーザ・ガイド*』および『*AN1086：Silicon Labs Bluetooth アプリケーションでの Gecko ブートローダの使用*』を参照してください。

7.3 RAM

Bluetooth スタックは Wireless Gecko の RAM の一部を予約し、アプリケーションのために未使用の RAM を確保しています。

Bluetooth 機能による RAM の消費は、以下のように分けられます。

- Bluetooth スタック
- Bluetooth 接続プール
- Bluetooth GATT データベース
- C STACK
- C HEAP

以下の表に RAM 使用量の詳細を示します。以下の数値は、ほとんどの機能が有効化されている一般的な使用例で示され、測定されています。お客様は、使用していない機能を無効にし、構成を変更することで、RAM の使用率を最適化できます。

コンポーネント	割り当てられた RAM
Bluetooth スタック	7 kB
Bluetooth 接続プール	4824 + 接続数 * 472 バイト
Bluetooth GATT データベース	アプリケーションに依存 (20～200 バイト)
呼び出しスタック	2 kB
ヒープ・メモリ	3 kB

7.3.1 Bluetooth スタック

Bluetooth スタックには 7 kB 以上の RAM が必要です。これには、低レベル無線ドライバを組み込んだ Bluetooth スタック・ソフトウェアとアプリケーション・プログラミング・インターフェイスが含まれます。

7.3.2 Bluetooth 接続プール

Bluetooth スタックは、動的メモリ割り当てにスタックの静的メモリ・プールを使用します。割り当てられるメモリ・プールのサイズは、並列接続の数によって異なります。この数は、`gecko_init()` 関数の `.bluetooth.max_connections` パラメータで設定します。

*Bluetooth 接続プール・サイズ = 4824 + 接続数 * 436 バイト*

7.3.3 Bluetooth GATT データベース

Bluetooth GATT データベースは RAM を使用します。使用される RAM の容量は、ユーザ定義の GATT データベースにより異なり、一般化できません。書き込み可能なすべての特性は、定義された長さに応じた容量の RAM を使用します。さらに、GATT 内の各属性には、属性の権限を維持するために、数バイトの RAM が必要です。一般的な RAM 使用量は約 20 ～ 200 バイトです。

7.3.4 呼び出しスタック

Bluetooth スタックには、呼び出しスタック用に 1.5 kB 以上の RAM が予約されている必要があります。アプリケーションの開発者は、スタックに必要な 1.5 kB に加え、アプリケーションの呼び出しスタック用に RAM を割り当てる必要があります。

呼び出しスタックのサイズ定義の位置は、コンパイラとスタートアップ・ファイルによって異なります。デフォルトの呼び出しスタック・サイズは 2 kB です。これは、以下のコマンド・ライン・オプションでオーバーライドできます。

コンパイラ	コマンド・ライン・オプション	注意
IAR	`--config_def __STACK_SIZE=`*`<size>`*	呼び出しスタックはリンカ・ファイルで定義されます。このパラメータをリンカに渡す必要があります。
GCC	`-D __STACK_SIZE=`*`<size>`*	呼び出しスタックはスタートアップ・コードで定義されます。これは、コンパイラに合わせて定義する必要があります。

7.3.5 ヒープ・メモリ

ヒープ・メモリはアプリケーション要件に基づいて予約する必要があります。Bluetooth スタックは、たいていの場合、Bluetooth のペアリング中に楕円曲線アルゴリズムを使用し、非対称暗号化演算のためのヒープ・メモリを必要とします。さらに、Bluetooth スタックは、一部のヒープ・メモリを使用して、コンポーネント間の内部通信およびアプリケーションへの BGAPI イベントの送信も行います。

ヒープのサイズ定義の位置は、コンパイラとスタートアップ・ファイルによって異なります。最小のヒープ・サイズは 3328 (0xD00) バイトで、これはデフォルト値でもあります。これは、以下のコマンド・ライン・オプションでオーバーライドできます。

コンパイラ	コマンド・ライン・オプション	注意
IAR	`--config_def __HEAP_SIZE=<size>`	呼び出しスタックはリンカ・ファイルで定義されます。このパラメータをリンカに渡す必要があります。
GCC	`-D__HEAP_SIZE=<size>`	呼び出しスタックはスタートアップ・コードで定義されます。これは、コンパイラに合わせて定義する必要があります。

第 8 章 アプリケーション ELF ファイル

ELF (Executable and Linkable Format) は、実行可能ファイルの標準ファイル形式です。この章では、アプリケーションおよび Bluetooth スタック に関連する、ELF ファイルの各セクションについて説明します。

一部のリンカは消費されるフラッシュを記述する出力を提供しますが、何が含まれているかは明らかではありません。Bluetooth プロジェクトにはブートローダと Bluetooth AppLoader が含まれることがあり、そのデバイスにはブートローダ用の個別のフラッシュが含まれる場合があります。ELF ファイルは、RAM およびフラッシュ使用量に関する正確な情報を提供します。

Simplicity Studio は、コマンド・ライン・ツール *objdump* を含む、GCC ツールチェーンを提供します。このツールを使用すると、ELF ファイルからセクション情報を入手できます。

objdump には入力 ELF ファイルが必要です。パラメータ -h を使用すると、*objdump* はセクションのヘッダ情報をダンプします。

IAR

サンプル・アプリケーションのコマンド・ラインから *objdump* を呼び出します。

```
arm-none-eabi-objdump -h IAR¥ ARM¥ -¥ Default/soc-thermometer-iar-mg1p.out
```

次に *objdump* は以下を出力します。

```
Sections: Idx Name Size VMA LMA File off Algn 0 .text_apploader rw 00008fc0 00004000 00004000 00000034 2**11 CONTENTS, ALLOC, LOAD, READONLY, DATA 1 .text_application us 0001e3d3 0000d000 0000d000 00008ff4 2**11 CONTENTS, ALLOC, LOAD, READONLY, CODE 2 A1 rw 00000800 20000000 20000000 000273c8 2**3 ALLOC 3 P3 rw 00000246 20000800 20000800 000273c8 2**2 ALLOC, CODE 4 P3 ui 00000d00 20000a48 20000a48 000273c8 2**3 ALLOC 5 P3 zi 00002b60 20001748 20001748 000273c8 2**8
```

.text_apploader は Bluetooth AppLoader を含みます。

.text_application は、アプリケーション・コードと読み取り専用データを含みます。この例でのアプリケーションのサイズは、16 進法で 0x1e3d3 バイト、10 進法で 123859 バイトです。

その他のセクションの記述については、IAR ドキュメントを参照してください。

GCC

サンプル・アプリケーションのコマンド・ラインから *objdump* を呼び出します。

```
arm-none-eabi-objdump -h GNU¥ ARM¥ v4.9.3¥ -¥ Default/soc-thermometer-gcc-mg1p.axf
```

次に *objdump* は以下を出力します。

```
Sections: Idx Name Size VMA LMA File off Algn 0 .text_bootloader 00000000 00000000 00000000 000306d0 2**0 CONTENTS 1 .text_apploader 00009000 00004000 00004000 00004000 2**0 CONTENTS, ALLOC, LOAD, READONLY, DATA 2 .text_application 0001e4c4 0000d000 0000d000 0000d000 2**8 CONTENTS, ALLOC, LOAD, READONLY, CODE 3 .text_application_ARM.exidx 00000008 0002b4c4 0002b4c4 0002b4c4 2**2 CONTENTS, ALLOC, LOAD, READONLY, DATA 4 .stack_dummy 00000400 20000000 20000000 000306d0 2**3 CONTENTS 5 .text_application_data 000002d0 20000400 0002b4cc 00030400 2**2 CONTENTS, ALLOC, LOAD, CODE 6 .bss 00002a88 20000700 0002b800 00030700 2**8 ALLOC 7 .heap 00000c00 20003188 20003188 00030ad0 2**3 CONTENTS
```

.text_bootloader にはブートローダが含まれます。この例では個別にロードされ、セクションは空になっています。

.text_apploader は Bluetooth AppLoader を含みます。

.text_application は、アプリケーション・コードと読み取り専用データを含みます。この例のアプリケーション・サイズは、16 進法で 0x1e3c3、10 進法で 124100 バイトです。

.text_application_ARM.exidx はデバッグに使用します。

.stack_dummy は呼び出しスタックのプレースホルダ・セクションです。

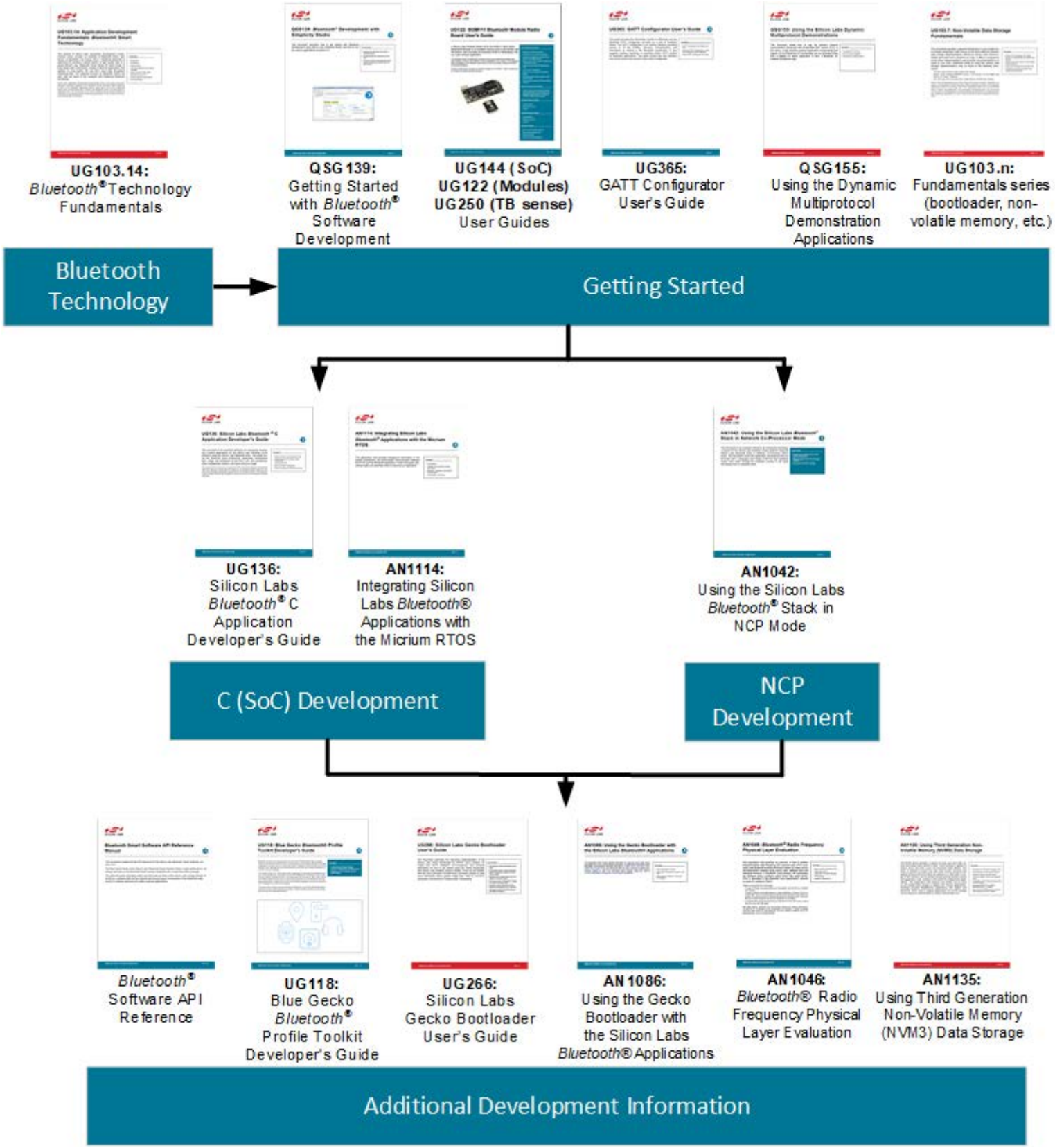
.text_application_data は初期化済み変数の RAM セクションです。

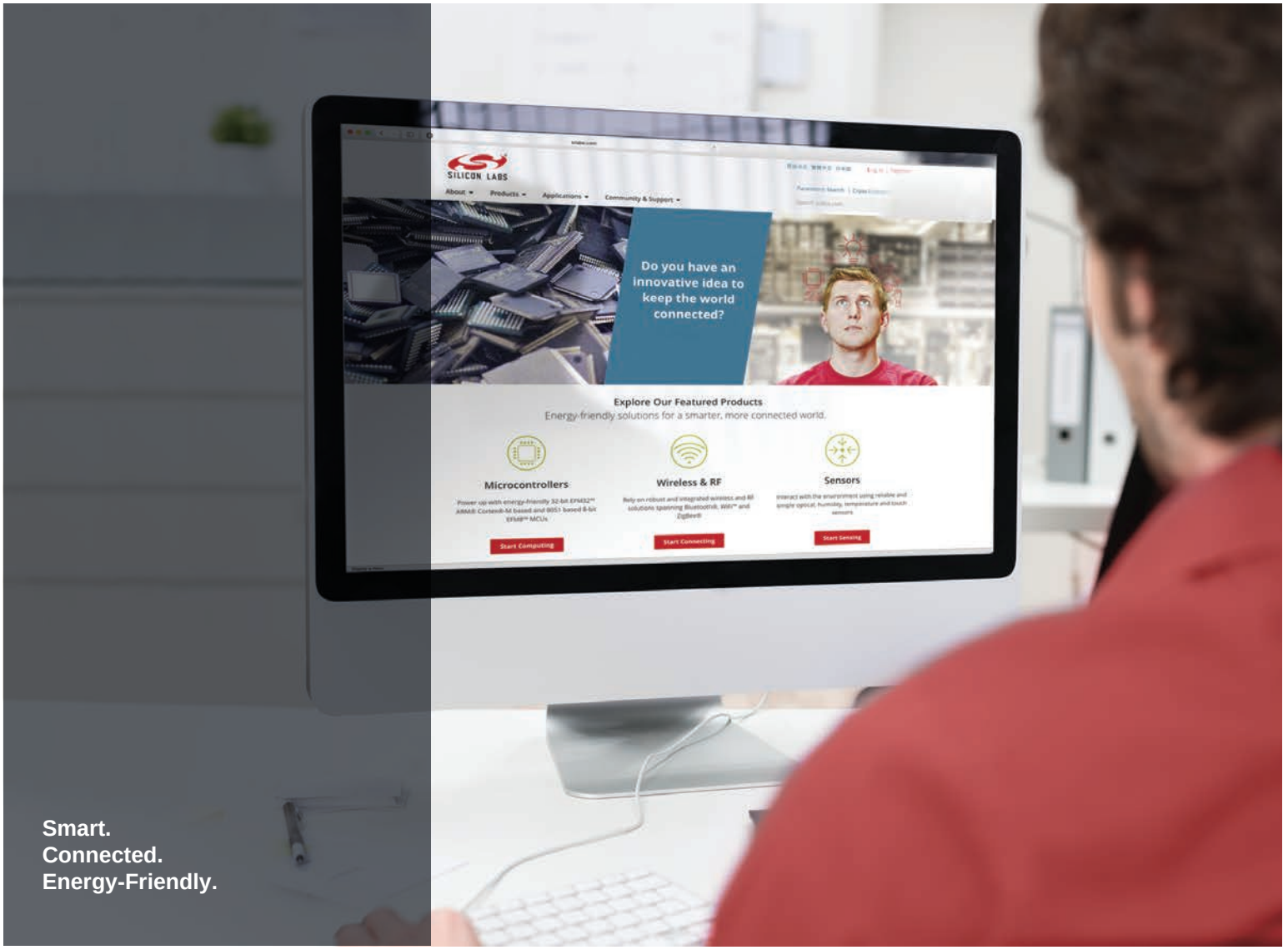
.bss は未初期化変数の RAM セクションです。

.heap はヒープの RAM セクションです。

その他のセクションの記述については、GCC ドキュメントを参照してください。

第 9 章 資料


UG103.14:
Bluetooth® Technology Fundamentals
QSG139:
Getting Started with Bluetooth® Software Development
UG144 (SoC)
UG122 (Modules)
UG250 (TB sense)
User Guides
UG365:
GATT Configurator User's Guide
QSG155:
Using the Dynamic Multiprotocol Demonstration Applications
UG103.n:
Fundamentals series (bootloader, non-volatile memory, etc.)
Bluetooth Technology
Getting Started
UG136:
Silicon Labs Bluetooth® C Application Developer's Guide
AN1114:
Integrating Silicon Labs Bluetooth® Applications with the Micrium RTOS
AN1042:
Using the Silicon Labs Bluetooth® Stack in NCP Mode
C (SoC) Development
NCP Development
Bluetooth® Software API Reference
UG118:
Blue Gecko Bluetooth® Profile Toolkit Developer's Guide
UG266:
Silicon Labs Gecko Bootloader User's Guide
AN1086:
Using the Gecko Bootloader with the Silicon Labs Bluetooth® Applications
AN1046:
Bluetooth® Radio Frequency Physical Layer Evaluation
AN1135:
Using Third Generation Non-Volatile Memory (NVM3) Data Storage
Additional Development Information



Products
www.silabs.com/products

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com