PSoC 6 MCU デバイス ファームウェア アップデート ソフトウェア開発キット ガイド

About this document

Scope and purpose

本アプリケーション ノート (AN213924) は、サイプレス PSoC 6 MCU 製品向けデバイス ファームウェア アップデート(DFU)システムを開発するための DFU ソフトウェア開発キット (SDK) の使用に関する包括的な情報を提供します。本書には、SDK アプリケーション プログラミング インターフェース (API) の詳細な説明および複数のブートローダ サンプル コードのビルド手順が記載されています。

関連製品ファミリ

すべての PSoC 6® MCU 製品

関連サンプル コード

関連資料をご参照ください

関連アプリケーション ノート

関連資料をご参照ください

その他のサンプル コードが必要な場合は、以下のとおりご対応いたします。

PSoC サンプル コードのリストにアクセスするには、**サンプル コードのウェブページ**をご覧ください。
サイプレス ビデオ トレーニング ライブラリについては、**ここ**からご覧ください。

Table of contents

About this document **1**
Table of contents **1**
1 はじめに **3**
2 デバイス ファームウェア アップデートとは? **4**
2.1 用語および定義 4
2.2 DFU モジュールの使用 5
2.3 基本的な DFU 機能フロー 5
2.4 その他のユースケース 7
3 DFU SDK の説明 **9**
3.1 DFU SDK ファイル 9
3.1.1 ユーザー コールバック関数 10
3.1.2 リンカースクリプト 11
3.2 DFU エコシステム 11
4 SDK の使用方法 **13**
4.1 システムでアプリケーションを決定 13
4.2 メモリにアプリケーションを配置 14

Table of contents

4.3 アプリケーションを設計 ... 15
4.3.1 ModusToolbox 手順 ... 15
4.3.2 PSoC Creator 手順 ... 19
4.4 アプリケーションのビルドとプログラム ... 25
5 PSoC 6 MCU DFU サンプル コード ... 27
5.1 サンプル コードのビルド方法 ... 28
5.2 PSoC 6 MCU の基本 DFU ... 28
5.2.1 ModusToolbox の手順 ... 29
5.2.2 PSoC Creator の手順 ... 32
5.3 PSoC 6 MCU BLE のブートローダ ... 38
5.3.1 アップグレードスタック付き BLE ブートローダ ... 44
5.4 PSoC 6 MCU のデュアルアプリケーション ブートローダ ... 46
5.5 PSoC 6 MCU 暗号化ブートローダ ... 48
6 関連資料 ... 52
7 付録 A. DFU ホスト ツール ... 54
8 付録 B. ホスト コマンド／応答プロトコル ... 56
8.1 コマンド／応答パケットの構造 ... 56
8.2 コマンド ... 56
8.2.1 DFU 開始 ... 57
8.2.2 DFU 同期化 ... 57
8.2.3 DFU 終了 ... 58
8.2.4 データ送信 ... 58
8.2.5 応答無しのデータ送信 ... 58
8.2.6 データ プログラミング ... 59
8.2.7 データ検証 ... 59
8.2.8 データ消去 ... 60
8.2.9 アプリケーション検証 ... 60
8.2.10 アプリケーション メタデータ設定 ... 61
8.2.11 メタデータ取得 ... 61
8.2.12 EIVector 設定 ... 62
9 付録 C. .cyacd2 ファイル フォーマット ... 63
10 付録 D. アプリケーション メタデータ ... 64
10.1 メタデータの構造 ... 66
11 付録 E. ビルド後のファイル一覧 ... 67
11.1 ModusToolbox のビルド後の Bash ファイル ... 67
11.2 PSoC Creator のビルド後のバッチ ファイル ... 68
改訂履歴 ... 70

1 はじめに

本書には、デバイスファームウェアアップデート (DFU, または"ブートローダ"と呼ばれる) 基礎の概要、サイプレスの DFU ソフトウェア開発キット (SDK) およびそれを PSoC 6 MCU に使用する方法の詳細な説明を提供します。

Note: *"ブートローダ"という用語は産業界で多重定義になっており、デバイスの起動 ("ブートアップ") プロセスと混同されることがよくあります。したがって、このアプリケーションノートと関連するサンプル コードは、あいまいさを減らすために"デバイス ファームウェア アップデート", または DFU に変えています。*

このガイドでは、サイプレスの ModusToolbox™統合開発環境 (IDE) やサイプレスの PSoC Creator IDE を含む、複数の開発ツールについて説明しています。ModusToolbox IDE は、Windows®, macOS®, および Linux® 用の無料の IDE です。PSoC 6 MCU と産業界で広く展開されている WiFi および Bluetooth 技術を組み合わせて、モノのインターネット (IoT) 設計者に単一で一貫性のある使い慣れた設計経験を提供します。ModusToolbox IDE の詳細については、ここをクリックしてください。

PSoC Creator は、複数のサイプレス PSoC MCU ファミリをベースとしたシステムのハードウェアとファームウェアの同時設計を可能にする無償の Windows ベースの IDE です。PSoC Creator の詳細については、ここをクリックしてください。

一般的な DFU / ブートローダに詳しくない方のために、基本的なコンセプトおよび設計原理を次の節「**デバイス ファームウェア アップデートとは?**」で説明します。

DFU の概念に精通し、DFU を PSoC 6 MCU に実装する方法を理解したいのであれば、「**DFU SDK の説明**」、「**SDK の使用方法**」、および「**PSoC 6 MCU DFU サンプル コード**」の節を参照してください。DFU のサンプル コード一覧については、「**関連資料**」を参照してください。PSoC 6 MCU サンプル コードの完全な一覧については、ここをクリックしてください。

Note: *公開時点で、UART、I^2C、SPI、および BLE ベースの DFU システムがサポートされています。他の通信チャネルは、今後の SDK アップデートで追加される予定です。*

2 デバイス ファームウェア アップデートとは?

デバイス ファームウェア アップデートは MCU システム設計で一般的に使用されます。DFU により、製品のファームウェアを現場で更新できます。一般的な製品で、ファームウェアは MCU のフラッシュ メモリに格納されています。**Figure 1** に示すように、MCU はプリント回路基板 (PCB) に取り付けられており、製品に組み込まれています。

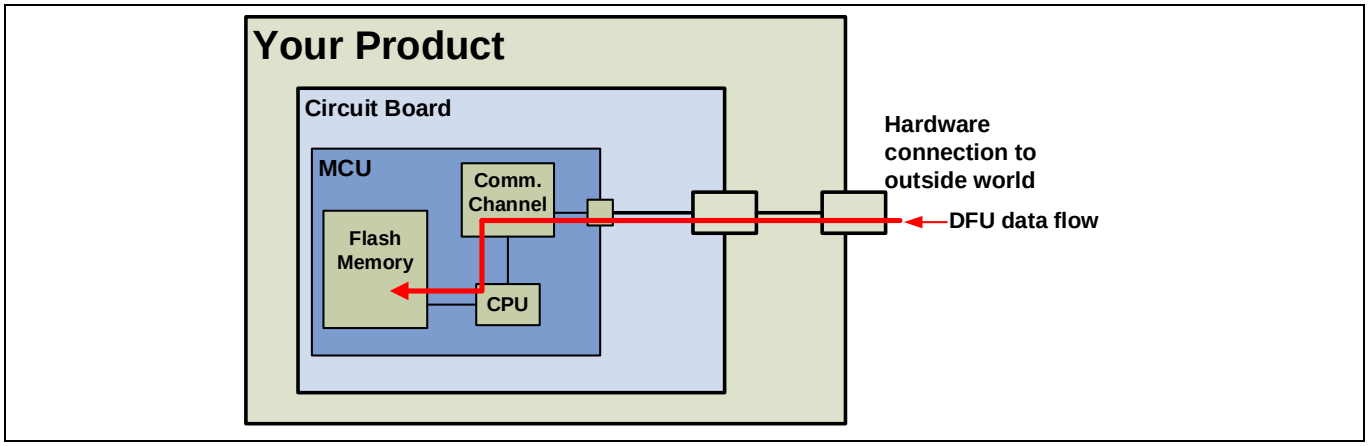


Figure 1 DFU データ フロー ブロックダイヤグラム

工場では、製品へのファームウェアの初期プログラミングは一般的に MCU の JTAG または SWD インターフェースを使用して PCB 組立時に行われます。しかし、それらのインターフェースは現場で常に利用できるわけではないため、一般的にはファームウェア更新に使用されません。

現場でファームウェアを更新するより良い方法は、製品と外部のデバイス間の既存の接続を利用することです。接続は、I^2C や USB、UART などの標準の通信ポート、BLE などのワイヤレス チャネル、またはカスタムのプロトコルです。

2.1 用語および定義

Figure 1 は、製品に組み込まれたファームウェアが、通常動作とフラッシュ更新という 2 つの異なる目的で、通信ポートを使用できなくてはならないことを示します。**Figure 2** に示すように、フラッシュの更新方法を知っている組込みファームウェアの部分は DFU モジュール、またはブートローダと呼ばれます。

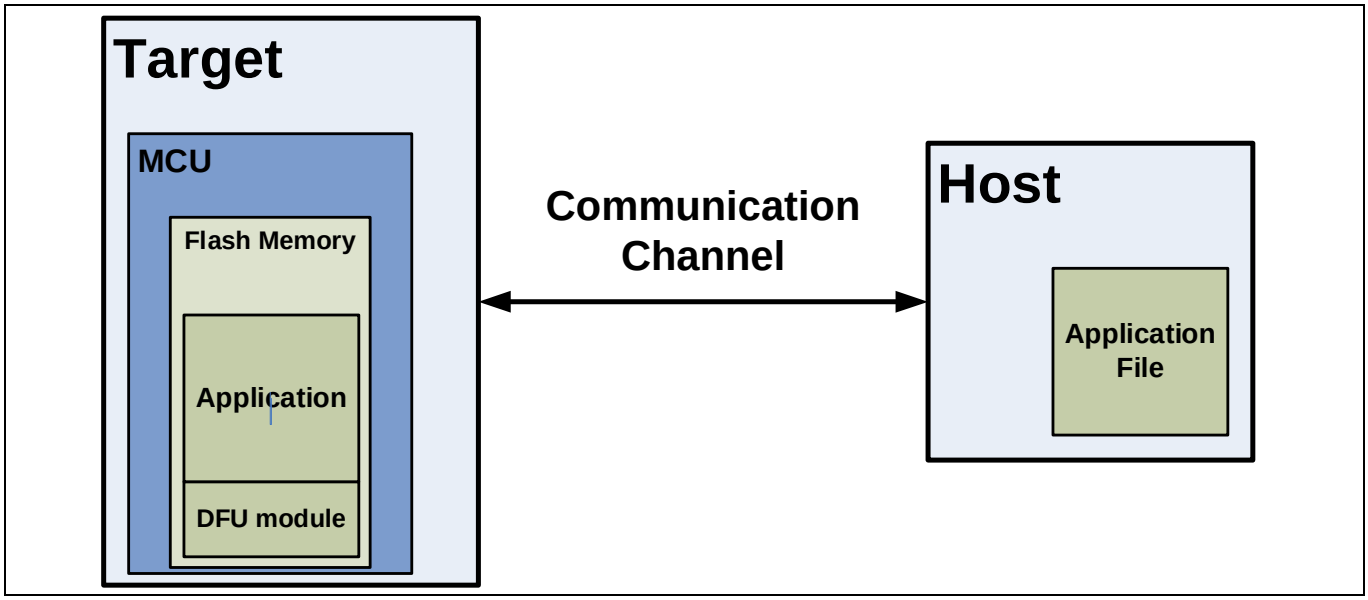


Figure 2　一般的な DFU システム

一般に、フラッシュ メモリを更新するデータを提供するシステムは**ホスト**、更新されるシステムは **ターゲット**と呼ばれます。ホストは外部コンピュータまたはターゲットと同じ PCB にある別の MCU です。

ホストからターゲットフラッシュにデータを転送する行為は、**DFU** または **DFU** 操作と呼ばれます (以前のドキュメントでは、サイプレスは DFU をブートローダと呼んでいました。しかし、この用語はしばしばブートアップ プロセスと混同されます。よって、このアプリケーションノートおよび関連するサンプル コードでは、あいまいさを減らすためにこれを DFU と呼びます。) フラッシュにあるデータは**アプリケーション**または**ファームウェア イメージ**と呼ばれます。

2.2　DFU モジュールの使用

通常、DFU モジュールとアプリケーションは通信ポートを共有します。DFU の最初のステップは、アプリケーションではなく DFU モジュールが実行されるモードに製品を配置することです。これは、ボタン押下などのイベントや受信されたコマンドに応答して行われます。アプリケーションはそのようなイベントを検出し、制御を DFU モジュールに渡すことで応答します。

DFU モジュールの実行後、ホストは通信チャネルを経由して「DFU の開始」コマンドを送信できます。DFU モジュールが「OK」応答を送信すると、DFU 操作を開始できます。

2.3　基本的な DFU 機能フロー

DFU 中、ホストは新しいアプリケーション用のファイルを読み出し、それを解析してフラッシュ書き込みコマンドにし、それらのコマンドを DFU モジュールに送信します。ファイル全体が受信され、ターゲット フラッシュにインストールされた後、DFU モジュールは新しいアプリケーションに制御を渡せます。

デバイス ファームウェア アップデートとは?

通常、DFU モジュールはデバイス リセット後にすぐに実行されます。[1]その後、以下の動作を行えます。

- 新しいアプリケーションに制御を渡す前に、そのアプリケーションの有効性を確認
- ホストとの通信を開始するタイミングを管理
- DFU 操作を実行
- 新しいアプリケーションに制御を渡す

Figure 3 はこの動作を示すフロー図です。

[1] 標準の MCU では、デバイスの電源投入やリセット ピンでの電圧レベルなどいくつかのイベントがデバイス リセットを起こします。また、ファームウェアはデバイス レジスタに書き込むことでリセットをトリガできます。これは「ソフトウェア リセット」(SRES) と知られています。SRES 機能により、アプリケーションはボタン押下などのイベントに応答してデバイスをリセットできます。DFU モジュールがデバイス リセット後にすぐに実行されるため、これは効果的に DFU モジュールに制御を渡します。

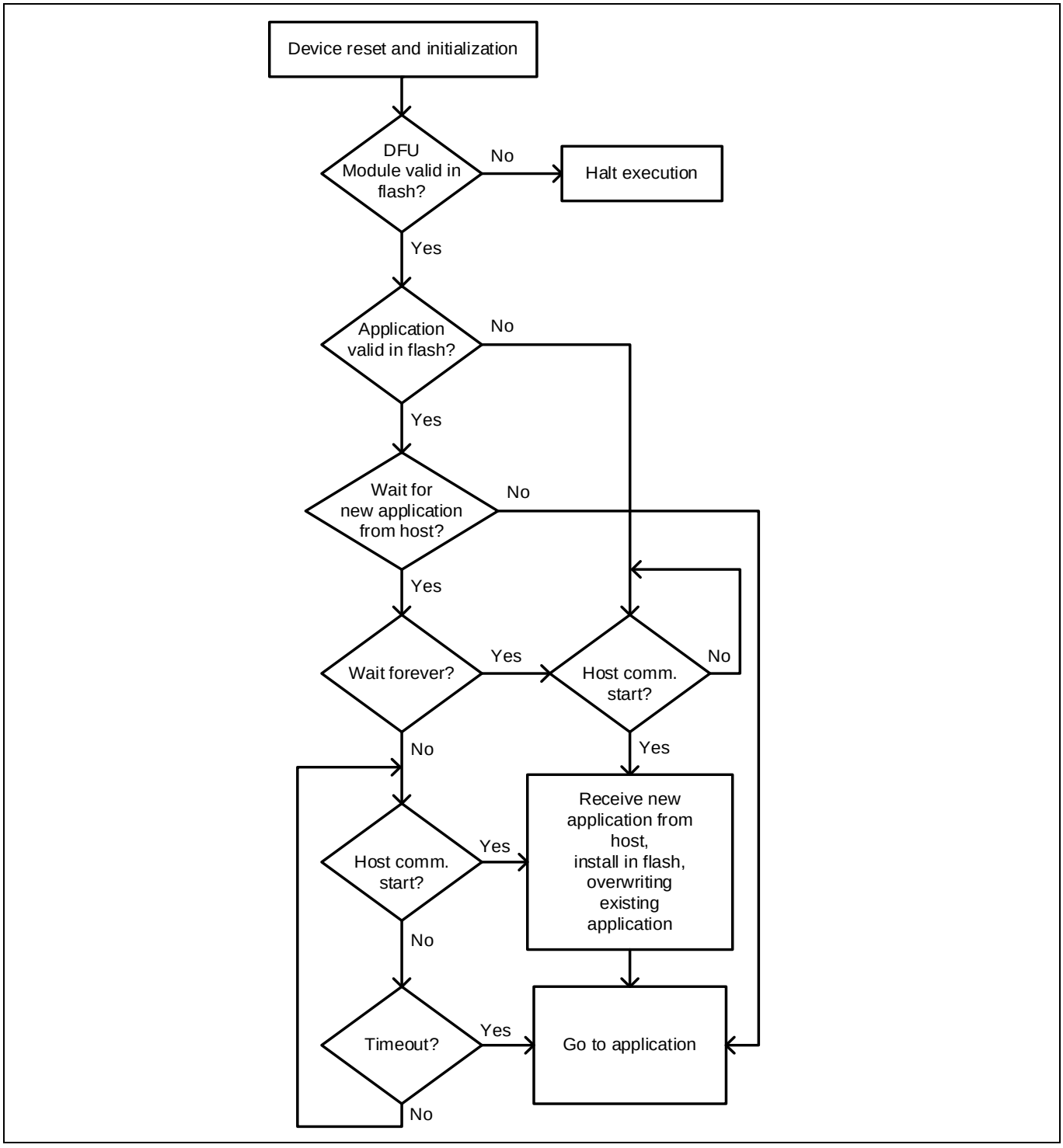


Figure 3　　DFU 機能フロー

2.4　　その他のユースケース

その他に、より複雑な DFU ユースケースがあります。一般的なユースケースとしては、アプリケーションを実行し、アプリケーションに利用可能な更新があることを通知するいくつかのイベントを作成します。

デバイス ファームウェア アップデートとは?

アプリケーションは通常のタスクを実行し続けながら、新しいアプリケーションを一時的な場所にダウンロードします。新しいアプリケーションの有効性が確認された後、フラッシュ内の正しい場所にコピーされ、制御権がそれに渡されます。

3 DFU SDK の説明

サイプレスの DFU SDK は、迅速な DFU システム開発を可能にする呼び出し可能関数の一式およびその他の要素から成る API です。SDK はサイプレス PSoC 6 SDK に同梱されています。SDK は、通常は *C:\ ... \ ModusToolbox_1.0 \ libraries \ psoc6sw-1.0 \ components \ psoc6mw \ dfu* に保存されています。SDK は他の IDE または ModusToolbox IDE によって使用できます。

SDK には以下のものが含まれます。

- SDK API を実装するソース コード (通常は.*h* および.*c* ファイルと、リンカー スクリプトおよび.*mk* ファイル)
- ドキュメント、すなわち、本書。
- サンプル コード

Note: *ホスト インターフェース プロトコルのコマンドの修正や追加などのカスタム目的で DFU SDK ソース コードを修正できます(「**付録 B: ホスト コマンド／応答プロトコル**」を参照してください)。*

3.1 DFU SDK ファイル

Table 1 で、SDK のファイルをリストアップします。

Note: *PSoC 6 MCU は Cortex®-M4 (CM4) と Cortex-M0+ (CM0+) の 2 個の Arm® CPU を内蔵しています。リンカー スクリプト ファイルは、複数の IDE に対応するようそれぞれの CPU 用に用意されています。詳細は、「**AN215656**, PSoC 6 MCU Dual-CPU System Design」を参照してください。*

Table 1 DFU SDK ファイル

ファイル	説明	
cy_dfu.h、*.c*	DFU SDK ファイル	
cy_dfu_bwc_macro.h	サイプレス レガシー ブートローダ SDK との後方互換性を実現	
dfu_user.h	SDK の動作および有効にされた機能を制御するユーザー編集可能な`#define`ステートメントを含む	
dfu_user.c	SDK に必要なユーザー関数を含み: DFU ホストとの通信を制御する 5 つの関数。「トランスポート関数」とも呼ばれる 内部または外部メモリへのアクセスを制御する `ReadData()` と `WriteData()` の 2 つの関数 提供される関数はデフォルトであり、アプリケーションに合わせて変更可能	
transport_uart.h、*.c* *transport_... .h*、*.c*	使用されるホスト通信チャネル用の DFU トランスポート関数を含む。これらは通常、*dfu_user.c* 内のトランスポート関数によって呼び出される	
dfu_cm4.ld、*dfu_cm0p.ld*	カスタムの GCC リンカー スクリプト	各アプリケーションでは、これらのリンカースクリプトファイルがデフォルトのリンカースクリプトファイルを置き換えます。これらのファイルは、DFU モジュールおよびその他の領域と同様に、各 CPU のコードおよびデータセクションを見つけます。
dfu_cm4.scat、*dfu_cm0p.scat*、*dfu_mdk_common.h*、*dfu_mdk_symbols.c*	カスタムの MDK リンカー スクリプト。 必要なシンボル定義を作成するための共通ファイルが存在します。	

ファイル	説明	
dfu_cm4.icf、*dfu_cm0p.icf*	カスタムの IAR リンカー スクリプト	

上記のファイルは、**Figure 4** に示すように DFU フォルダ内で構成されています。

```
\---dfu
    |   cy_dfu.h
    |   cy_dfu.c
    |   cy_dfu_bwc_macro.h
    \---config
    |       dfu_user.h
    |       dfu_user.c
    |       transport_uart.h
    |       transport_uart.c
    |       transport_... .h  transport files for other
    |       transport_... .c  communication channels
    \---linker_scripts
        \---GCC
        |       dfu_cm0p.ld
        |       dfu_cm4.ld
        \---MDK
        |       dfu_cm0p.scat
        |       dfu_cm4.scat
        |       dfu_mdk_common.h
        |       dfu_mdk_symbols.c
        \---IAR
                dfu_cm0p.icf
                dfu_cm4.icf
```

Figure 4　DFU SDK ファイルの構成

3.1.1　ユーザー コールバック関数

DFU SDK API 関数はいくつかのユーザー関数をコールバックします。これにより、以下の DFU 動作をカスタマイズできます。

- ホストとの通信。トランスポート関数または通信インターフェース関数とも呼ばれる
- デバイスの内部フラッシュおよび外部フラッシュなどその他の不揮発性メモリ (NVM) への読み書き

Table 2 で、必要なユーザー コールバック関数をリストアップします。これらの関数の標準コードを示す DFU サンプル コードについては、**PSoC 6 MCU DFU サンプル コード**および**関連資料**の節を参照してください。

Table 2　　ユーザー コールバック関数

関数	説明
ホストとの通信用。サンプル コードが dfu_user.c と *transport_xxx.h* と.c ファイルにある	
`Cy_DFU_TransportStart()`	通信チャネルを開き、初期化
`Cy_DFU_TransportStop()`	通信チャネルを閉じる
`Cy_DFU_TransportReset()`	チャネルを再初期化し、通常は既知の状態に戻るため
`Cy_DFU_TransportRead()`	ホストからパケットを受信。**ホスト パケット構造**を参照してください
`Cy_DFU_TransportWrite()`	ホストにパケットを送信。**ホスト パケット構造**を参照してください
不揮発性メモリ (NVM) アクセス用。サンプル コードが *dfu_user.c* にある	
`Cy_DFU_ReadData()`	デバイスのフラッシュまたは他の NVM からデータを読み出す
`Cy_DFU_WriteData()`	デバイスのフラッシュまたは他の NVM にデータを読書き込む

3.1.2　リンカースクリプト

DFU システムは、NVM に複数のアプリケーション (少なくとも 2 つ) があるシステムと定義されます。各アプリケーションが NVM のどこにあるかを決定します。DFU SDK には、ユーザーのアプリケーションに適用できるリンカー スクリプト ファイル (**Table 1** および **Figure 4** を参照してください) が含まれます。DFU サンプル コード、サンプル リンカー スクリプト ファイルが含まれます。

3.2　DFU エコシステム

DFU システムのその他の要素は SDK に含まれません。それらは ModusToolbox IDE, PSoC 6 SDK の他の場所, PSoC Creator IDE, またはペリフェラルドライバライブラリ (PDL) で利用可能で、他の IDE によって使用できます。

- CyMCUElfTool：*... \ ModusToolbox_1.0 \ tools \ cymcuelftool-1.0 \ bin* フォルダ内の *cymcuelftool.exe*。 このプログラムは、**Figure 5** に示すように、アプリケーション構築の一部として呼び出されます。(図は PSoC Creator および他の IDE に類似しています。)
- このプログラムは、コアとアプリケーションのイメージファイルを出力ファイルに結合します。これはコマンドラインプログラムです。オプションの構文は、ヘルプコマンド (`-h / --help`) の出力に記載されています。ユーザーガイドは*... \ cymcuelftool-1.0 \ doc* フォルダにあります。
- その他のドライバー (通信やフラッシュなどのため): ... \ ModusToolbox_1.0 \ libraries \ psoc6sw-1.0 \ components \ psoc6pdl \ drivers にあります。必要に応じて、DFU SDK の API 関数によって呼び出されます。
- デバイスファームウェア更新ホストツール：*... \ ModusToolbox_1.0 \ tools \ dfuh-tool-1.0* フォルダ内の *dfuh-tool.exe*。 これは、**Figure 2** に示す「ホスト」の例です。詳細については、**7「付録 A. DFU ホスト ツール」**を参照してください。

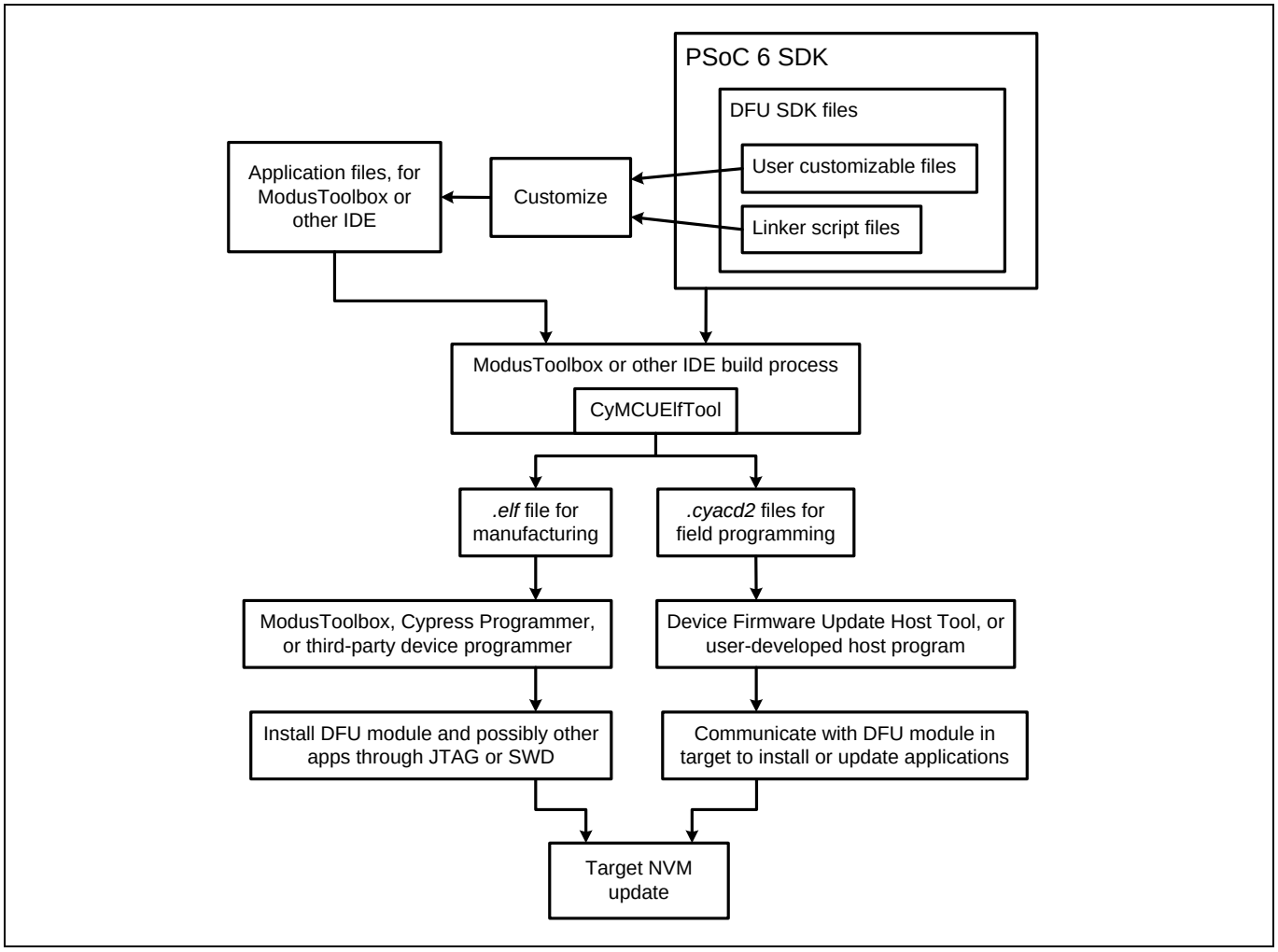


Figure 5　ModusToolbox アプリケーション ビルドおよび DFU プロセスダイヤグラム

4 SDK の使用方法

本節で、DFU SDK を使用するための一般的なプロセスを説明します。高レベルの手順は以下の通りです。

- **システムでアプリケーションを決定**
- **メモリにアプリケーションを配置**
- **アプリケーションを設計**
- **アプリケーションをビルドおよびプログラム**

DFU SDK のサンプル コードのビルド手順については、**PSoC 6 MCU DFU サンプル コード**を参照してください。

4.1 システムでアプリケーションを決定

最初のステップは、設計のアプリケーションの数およびそれぞれの目的を決定することです。どのアプリケーションに DFU 機能があるのでしょうか? **Figure 6** に複数のアプリケーションに対応したいくつかの標準的なメモリ マップを示します。

- マップ **A** は、アプリケーション#0 がアプリケーション#1 をダウンロードおよびインストールする基本的なケースを示します。
- マップ **B** は、より複雑なケースを示します。3 つのアプリケーションがあります。すべてのアプリケーションには DFU モジュールがあり、他のアプリケーションをダウンロードしてインストールできます。 (実際には、アプリケーション#0 は常に更新されません。) すべてのアプリケーションの外に一般的なデータ セクションが配置されています。
- マップ **C** は、アプリケーションの 1 つが外部不揮発性メモリ (NVM) に配置されることを除き、マップ **B** と同様なケースを示します。

それぞれのマップで、DFU システムはフラッシュの小さい部分 (通常は 1 行) を使用してアプリケーション メタデータを保存します。詳細は、**10 付録 D. アプリケーション メタデータ**を参照してください。

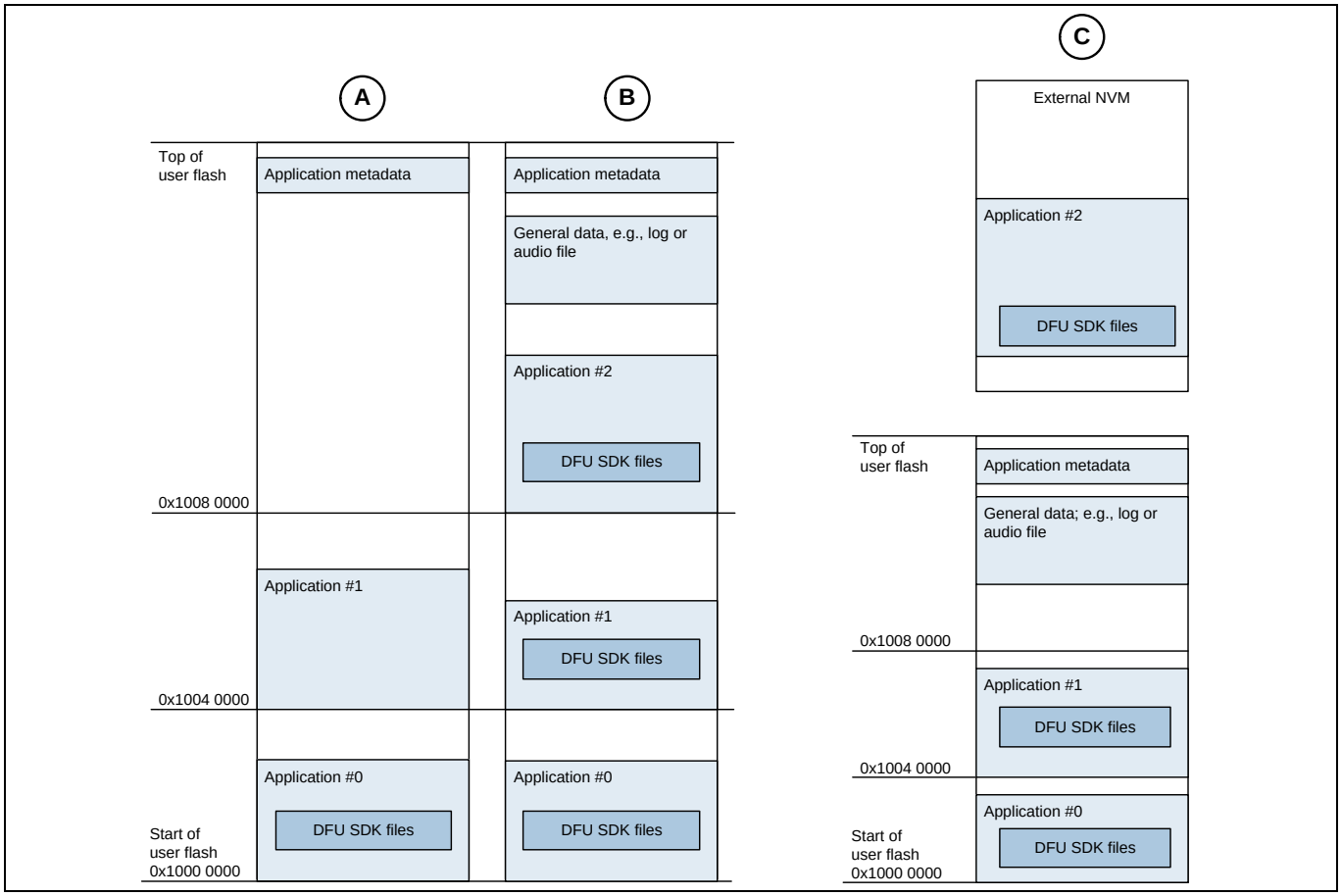


Figure 6　DFU のメモリ マップ例

4.2　メモリにアプリケーションを配置

次のステップとしては、フラッシュ内でどこにそれぞれのアプリケーションを配置するかを決定します。以下のような検討要素があります。

- メタデータ サイズに応じて、最大 63 のアプリケーションに対応しています。DFU SDK メタデータの詳細については、**10 付録 D. アプリケーション メタデータ**を参照してください。
- 1 つのアプリケーションが、1 つ以上の CPU 用のコードとデータを含む場合があります (例えば、PSoC 6 MCU デバイス内の CM4 と CM0+)。詳細は、「**AN215656**, *PSoC 6 MCU Dual-CPU System Design*」アプリケーション ノートを参照してください。
- 通常「App0」と呼ばれる最初のアプリケーションは、ユーザー フラッシュの先頭に配置しなければなりません (PSoC 6 MCU では、**Figure 6** に示すようにアドレス 0x1000 0000 です。) これは、デバイス リセット後に最初に実行するアプリケーションです。[1]

Note:　*セキュア システムにおけるセキュア イメージ モジュールなどの他のモジュールが 0x1000 0000 を占め、App0 がフラッシュ内の他の場所に配置される場合があります。詳細については、「**AN221111**, Creating a Secure System」を参照してください。*

1 CM0+ CPU はデバイス リセット後にすぐに SROM および他のシステム レベル コードを実行してから、0x1000 0000 に配置されたアプリケーションに制御を渡します。詳細については、デバイスの **Technical Reference Manual (TRM)** を参照してください。

- どのアプリケーションにも DFU SDK が組み込まれているため、他のアプリケーションをダウンロードしてインストールしたり、制御を他のアプリケーションに渡したりできます。ほとんどの場合、App0 には DFU SDK を組み込む必要があります。
- 非ブロックモードでフラッシュを書き込む場合、PSoC 6 MCU には 512 KB の領域にまたがる Read-while-Write (RWW) 機能があります。つまり、DFU コードは 1 つのフラッシュ領域で実行され、別の領域でフラッシュを更新します。詳細については、PDL フラッシュドライバーのドキュメントとデバイスの **Technical Reference Manual (TRM)** を参照してください。
- アプリケーションは **Figure 6** のマップ **C** に示すように、外部メモリにもデバイスの内部フラッシュにも配置できます。外部メモリは、0x1800 0000～0x1FFF FFFF の 128MB サブ領域にあります。

Note: *外部メモリの外で実行時、外部メモリ デバイスは XIP モードでの QSPI ブロックを使用して PSoC 6 MCU にマッピングしなければなりません。詳細については、デバイスの **Technical Reference Manual (TRM)** を参照してください。*

- PSoC 6 MCU のアプリケーションは、アプリケーションの一部として、NVM に保存された検証バイトを含みます。これは、制御権移転の前にアプリケーションが検証されることを可能にします。

Note: *アプリケーションが存在するセキュリティ環境に応じて、アプリケーションはシンプルな検証 (チェックサムまたは CRC データを使う) および認可 (暗号化されたシグネチャ データを使う) 向けに構成できます。DFU システムは検証を直接実施するか、またはフラッシュ ブート モジュール内で関数を呼び出して認可を確認します。詳細については、「**AN221111**, Creating a Secure System」を参照してください。*

- イベント ログなど通常のワークフローで更新される可能性がある NVM データは、アプリケーション検証か困難にならないように、アプリケーションの一部として含んではいけません。その代わりに、アプリケーションの空間の外にある NVM 領域を指定してデータ ストレージに使用します。例は、**Figure 6** に示したマップ **B** と **C** です。
- **Figure 6** に示すように、アプリケーション メタデータ用に 1 行のフラッシュ (まれに複数行が必要になる場合があります) を予約します。アプリケーション メタデータは DFU SDK によって管理され、アプリケーションが別のアプリケーションに制御を渡すために使用します。アプリケーション メタデータの空間はアプリケーション空間の外にある必要があります。

4.3 アプリケーションを設計

4.3.1 ModusToolbox 手順

Note: *ここでは、DFU 固有の手順を含みます。ModusToolbox アプリケーションを作成する手順については、ModusToolbox のヘルプ セクション、「**AN221774**, Getting Started with PSoC 6 MCU」または本書の「**PSoC 6 MCU DFU サンプル コード**」節を参照してください。*

次のステップは、**前ステップ**で特定した各アプリケーションを設計することです。それぞれのアプリケーションは、その他のアプリケーションから独立した単一の ModusToolbox アプリケーションです。ModusToolbox IDE では、すべてのアプリケーションを 1 つのワークスペース、または個別のワークスペースやコンピュータ内の個別の場所に配置できます。PSoC 6 MCU の開発を始める前に、システム開発全体のニーズに合わせてワークスペースとアプリケーションの計画の作成をお勧めします。

Note: *PSoC 6 MCU (デュアル CPU) 用の ModusToolbox アプリケーションは、5 つのプロジェクトで構成されています。*

- <project name>_config: デバイス構成ファイル design.modus、およびデバイス構成ソースファイル
- <project name>_mainapp: Cortex-M4 (CM4) CPU のソースファイル
- <project name>_mainapp_cm0p: Cortex-M0+ (CM0+) CPU のソースファイル
- <project name>_mainapp_psoc6pdl: Cortex-M4 (CM4) CPU のドライバーファイル
- <project name>_mainapp_cm0p_psoc6pdl: Cortex-M0+ (CM0+) CPU のドライバーファイル

DFU 操作またはインストール可能なアプリケーションとなる各 ModusToolbox アプリケーションに対して以下を行ってください。

1. アプリケーションが DFU 操作を行う場合、*design.modus* ファイルで通信周辺機器を有効にします。その周辺機器は、DFU ホストへの通信チャネルを実装します。

DFU SDK には、UART、SPI、I²C、BLE など、多くの通信チャネルタイプのサポートが含まれています。サンプル コードは、各チャネルタイプで利用できます。 **PSoC 6 MCU DFU サンプル コード**と**関連資料**を参照してください。

カスタム通信チャネルも作成できます。ドライバーは、**ユーザー コールバック関数**で説明されているトランスポート関数を実装する必要があります。

2. **Figure 7** に示すように、DFU SDK をプロジェクトに組み込みます。Project Explorer ウィンドウ内で _mainapp プロジェクトを右クリックし、**ModusToolbox Middleware Selector** をクリックします。

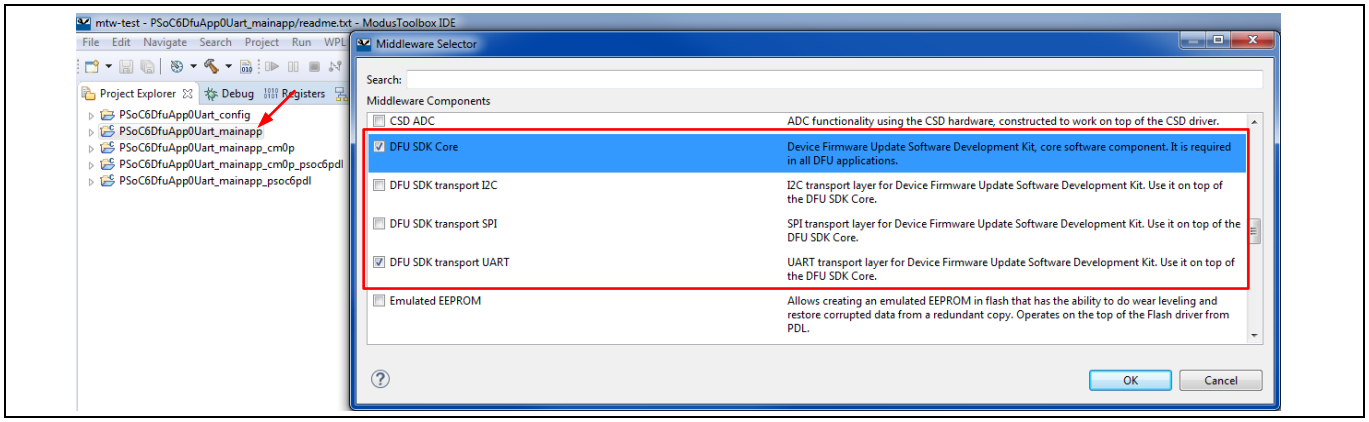


Figure 7 DFU SDK を ModusToolbox プロジェクトに組み込み

アプリケーションが DFU 操作を実行する場合は、**DFU SDK Core** および **DFU SDK transport xxx** ボックスを選択します。

Note: ほとんどの場合、1 つのトランスポート ボックスのみが選択されます。ただし、カスタム通信チャネルが必要な場合は、すべてのトランスポート ボックスを選択しないままにするか、複数のボックスの選択もできます。

アプリケーションがダウンロード可能なアプリケーションであり、制御を別のアプリケーションに転送する場合は、**DFU SDK Core** を選択します。制御を別のアプリケーションに転送するコードは、DFU SDK コア ファイルに含まれています。

完了したら **OK** をクリックします。必要なソース *.c, .h,* およびリンカースクリプト (例：*.ld*) ファイルがプロジェクトに自動的に追加されます。

_mainapp_cm0p プロジェクトに対してこの手順を繰り返します。ただし、**DFU SDK core** ボックスのみを選択します。

3. インストールされたリンカースクリプトファイルは、デフォルトでアプリケーション#0 (App0) 用に設定されています。他のアプリケーションの場合、アプリケーション番号を変更してファイルを編集します。次の例は、*dfu_cm0p.ld* および *dfu_cm4.ld* にある、GCC リンカーのダウンロード可能なアプリケーション App1 の編集を示しています。

```
/*
* DFU SDK specific: aliases regions, so the rest of the code does not use
* application-specific memory region names
*/
REGION_ALIAS("flash_core0", flash_app1_core0);
REGION_ALIAS("flash",       flash_app1_core1);
REGION_ALIAS("ram",           ram_app1_core1);


/* DFU SDK specific: sets an app Id */
__cy_app_id = 1;
```

4. プロジェクトプロパティを変更して、デフォルトのリンカースクリプトファイルの代わりに DFU リンカースクリプトファイルを使用します。**Figure 8** に示すように、_mainapp の *dfu_cm4.ld* ファイルを選択します。 ファイルへの相対パスを含めます。 _mainapp_cm0p の *dfu_cm0p.ld* ファイルを選択します (図には表示されていません)。

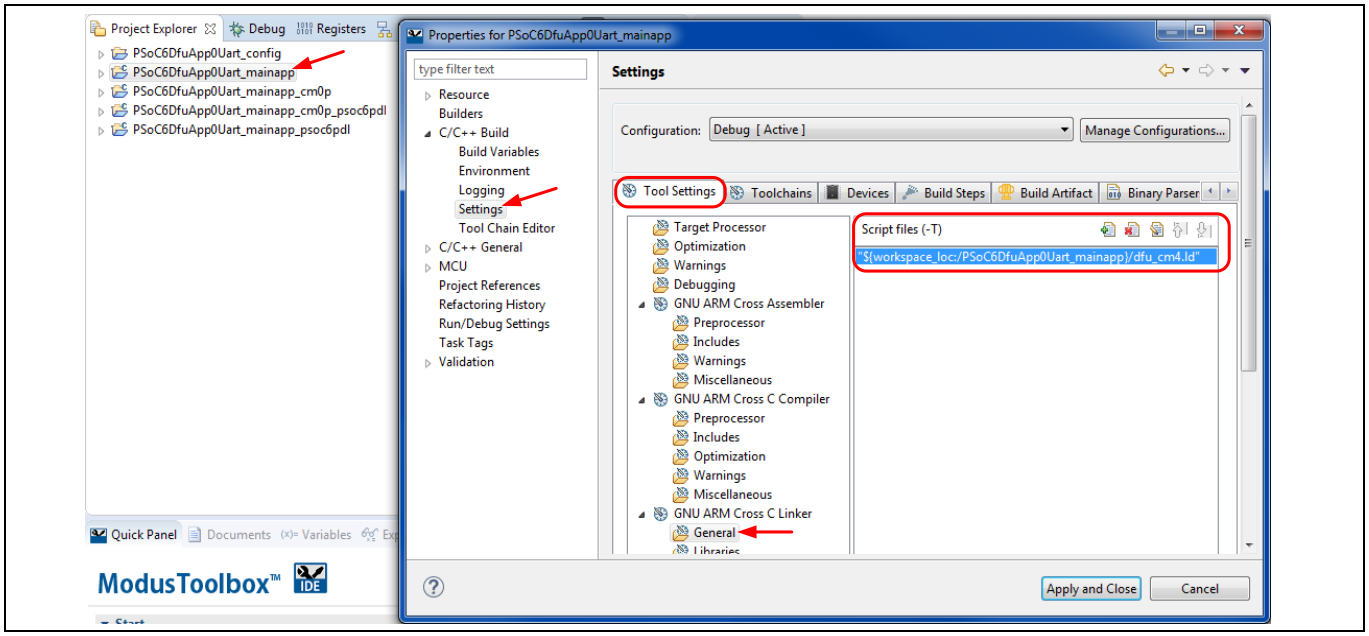


Figure 8　カスタムリンカースクリプトの ModusToolbox プロジェクトプロパティの設定

5. ダウンロード可能なアプリケーションプロジェクトの場合、**Figure 9** に示すように、ビルド後のコマンドを追加してサイプレス ユーティリティ プログラム cymcuelftool.exe を呼び出します。**cymcuelftool** は ModusToolbox のインストールに含まれています。DFU ホストによってダウンロードされる*.cyacd2 ファイルを生成します (**Figure 5** を参照)。ビルド後のコマンドは、CM4 バイナリに対してのみ実行されます。cymcuelftool は、CM4 (__mainapp) と CM0+ (mainapp_cm0p) プロジェクトを組み合わせて、出力.cyacd2 ファイルを作成します。

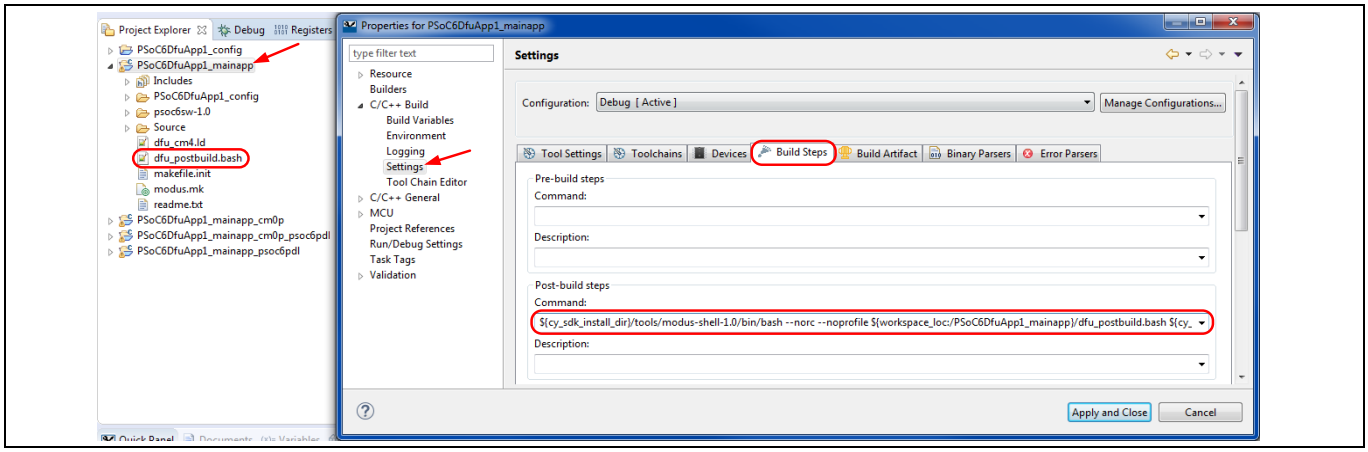


Figure 9 ModusToolbox のビルド後のコマンド

このコマンドは通常、サンプル コード **CE213903** の *dfu_postbuild.bash* などの bash ファイル、*PSoC 6 MCU Basic DFU* を呼び出します。便宜上、ファイルをコピーして_mainapp プロジェクトに貼り付けます。

次に、_mainapp プロジェクトのプロパティを変更して、**Figure 9** に示すように、次のようなコマンドテキストを追加します。

```
${cy_sdk_install_dir}/tools/modus-shell-1.0/bin/bash --norc --noprofile
${workspace_loc:/PSoC6DfuApp1_mainapp}/dfu_postbuild.bash
${cy_sdk_install_dir}/tools/cymcuelftool-1.0/bin/cymcuelftool
${workspace_loc:/PSoC6DfuApp1_mainapp_cm0p}/${config_name:PSoC6DfuApp1_mainapp_cm0p}/PSoC6DfuApp1_mainapp_cm0p.elf
${workspace_loc:/PSoC6DfuApp1_mainapp}/${config_name:PSoC6DfuApp1_mainapp}/PSoC6DfuApp1_mainapp.elf ARM_CM4
```

6. 必要に応じて、CM4 (__mainapp) および CM0+ (mainapp_cm0p) プロジェクトの両方の *main.c* およびその他のソースファイルにコードを追加します。 具体的な要件は次のとおりです。

- DFU SDK API にアクセスするために必要なすべてのソースファイルに `#include "cy_dfu.h"` ステートメントを追加します。
- ダウンロード可能なアプリケーションの場合、__ mainapp_cm0p プロジェクトの *main.c* で、`Cy_SysEnableCM4()`関数呼び出しを次のように変更します:
  `Cy_SysEnableCM4((uint32_t)(&__cy_app_core1_start_addr));`
- ダウンロード可能なアプリケーションの場合、__ mainapp プロジェクトの *main.c* で、次のグローバルステートメントを追加します。選択した署名タイプの配列サイズを調整します。

```
/* This section holds signature data for application verification. */
CY_SECTION(".cy_app_signature") __USED static const uint32_t
cy_bootload_appSignature[1];
```

両方のメインファイルのコード全体で、次のことを考慮してください。

- 呼び出す DFU 関数
  - `Cy_DFU_Complete()`: DFU 操作全体の実行中にブロックします。DFU 中に実行する他のタスクがない場合、この関数を呼び出します。

  - Cy_DFU_Init() の後に、Cy_DFU_Continue()への一連の呼び出しが続きます。Cy_DFU_Continue()は、ホストからの 1 つのコマンドパケットの受信, 処理, および応答中にブロックします。DFU 中に他のタスクを実行する必要がある場合は、これらの関数を呼び出します。
- 各アプリケーションが制御を別のアプリケーションに渡すかどうか。必要に応じて、DFU SDK API 関数 Cy_DFU_ValidateApp() および Cy_DFU_ExecuteApp() への呼び出しを追加します。

詳細については、PDL DFU SDK のドキュメントを参照してください。

Note: ***PSoC 6 MCU DFU サンプル コード***および***関連資料***にリストされているサンプル コードからコードをコピーして貼り付けられます。

4.3.2 PSoC Creator 手順

Note: ここでは、ブートローダ固有の (ModusToolbox IDE では、これは DFU 操作に変更) 手順を含みます。PSoC Creator プロジェクトを作成する手順については、PSoC Creator のヘルプ セクション、「***AN210781***, Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity」または本書の***サンプル コードのビルド方法***を参照してください。

次のステップは、**前ステップ**で特定した各アプリケーションを設計することです。それぞれのアプリケーションは、その他のアプリケーションから独立した単一の PSoC Creator プロジェクトです。PSoC Creator では、すべてのプロジェクトを 1 つのワークスペース (*.cywrk* ファイル)、または個別のワークスペースやコンピュータ内の個別の場所に配置できます。PSoC 6 MCU の開発を始める前に、システム開発全体のニーズに合わせてワークスペースとプロジェクトの計画の作成をお勧めします。

ブートローダまたはインストール可能なアプリケーションとなる各 PSoC creator プロジェクトに対して以下を行ってください。

1. ブートローディング向けのアプリケーションの場合、プロジェクト回路図 (*TopDesign.cysch* ファイル) に通信コンポーネントを配置します。このコンポーネントはブートローダホストへの通信チャネルを実装するものです。コンポーネントの端子を適切な物理ピンに接続します。

   ブートローダ SDK は、UART、SPI、I^2C、BLE を含む PSoC Creator コンポーネントカタログの多くの通信コンポーネントに対応しています。それぞれの通信コンポーネントのサンプルコードについては、**PSoC 6 MCU DFU サンプル コード**および**関連資料**を参照してください。

   Note: *SDK のデフォルトのトランスポートファイルとの互換性を確保するために、コンポーネントを **UART**、**I2C**、**SPI** または **BLE** と名付けます。*

   また、カスタムの通信チャネルを作成できます。それは、**ユーザー コールバック関数**で説明したトランスポート関数を実行しなければなりません。

2. **Figure 7** に示すように、ブートローダ SDK をプロジェクトに組み込みます。Workspace Explorer ウィンドウ内でプロジェクトを右クリックし、**Build Settings...**を選択します。**Build Settings** ダイアログで、**Peripheral Driver Library** を選択します。

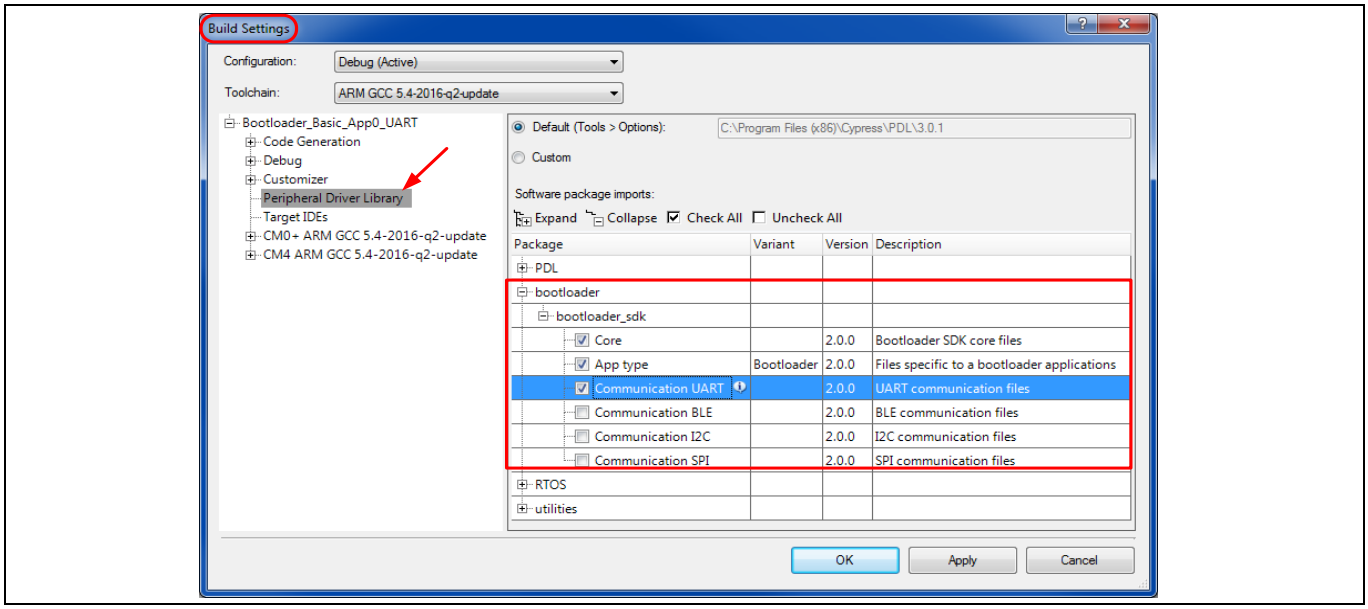


Figure 10　ブートローダ SDK を PSoC Creator プロジェクトに組み込み

アプリケーションがブートローダ操作を実行する場合は、**Core**, **App type bootloader,** および **Communication** ボックスを選択します。

Note: *ほとんどの場合、1 つのコミュニケーション ボックスのみが選択されます。ただし、カスタム通信チャネルが必要な場合は、すべてのコミュニケーション ボックスを選択しないままにするか、複数のボックスの選択もできます。*

アプリケーションがダウンロード可能なアプリケーションであり、制御を別のアプリケーションに転送する場合は、**Core** を選択します。制御を別のアプリケーションに転送するコードは、SDK コア ファイルに含まれています。

完了したら **OK** をクリックします。必要なソース.*c*, *.h*, およびリンカースクリプト (例：*.ld*) ファイルがプロジェクトに自動的に追加されます。

3. プロジェクトファイルを生成します。Workspace Explorer ウィンドウ内でプロジェクトをクリックし、**Build** > **Generate Application** を選択します。ファイルは、**Figure 11** に示すようにプロジェクトに追加されます。

   PSoC 6 MCU は CM4 と CM0+の 2 個の Arm CPU を内蔵していることに注意してください。それぞれの CPU はそれ自身のフォルダ、一般フォルダおよび共有ファイルフォルダを持っています。複数の IDE に対応するそれぞれの CPU のコードを特定するリンカースクリプトファイルは自動的にコピーされ、プロジェクトに追加されます。詳細は、「**AN215656**, *PSoC 6 MCU Dual-CPU System Design*」を参照してください。

Note: ***Figure 11*** *に、CM0p フォルダに追加されたブートローダファイルを示します。同様なファイルも CM4 フォルダに追加されますが、示されません。*

4. *bootload_user.h* と *bootload_user.c* ファイルを編集します。*bootload_user.c* 内の**ユーザー コールバック関数**を確認し、外部メモリへの読み書きなどの必要なカスタマイズのために編集します。通常は、関数は変更されずにそのままにされます。

Note: *追加された bootload_user.c ファイルは UART のデフォルト コードを含みます。I²C や SPI などその他の通信チャネルの場合、以下のように bootload_user.c を編集します。*

- 編集前:
  `#include "transport_uart.h"`
  編集後:
  `#include "transport_i2c.h"`
  または:
  `#include "transport_spi.h"`
  または:
  `#include "transport_ble.h"`

- “UART_Uart”の 5 つのインスタンスを“I2C_I2c”、“SPI_Spi”または“CyBLE”に変更します。

5. ご使用のコンパイラによって、適切な共通のリンカー スクリプト ファイルを編集し、「**メモリにアプリケーションを配置**」で行った決定をエンコードします。例は、**5.3 節のステップ 9** を参照してください。

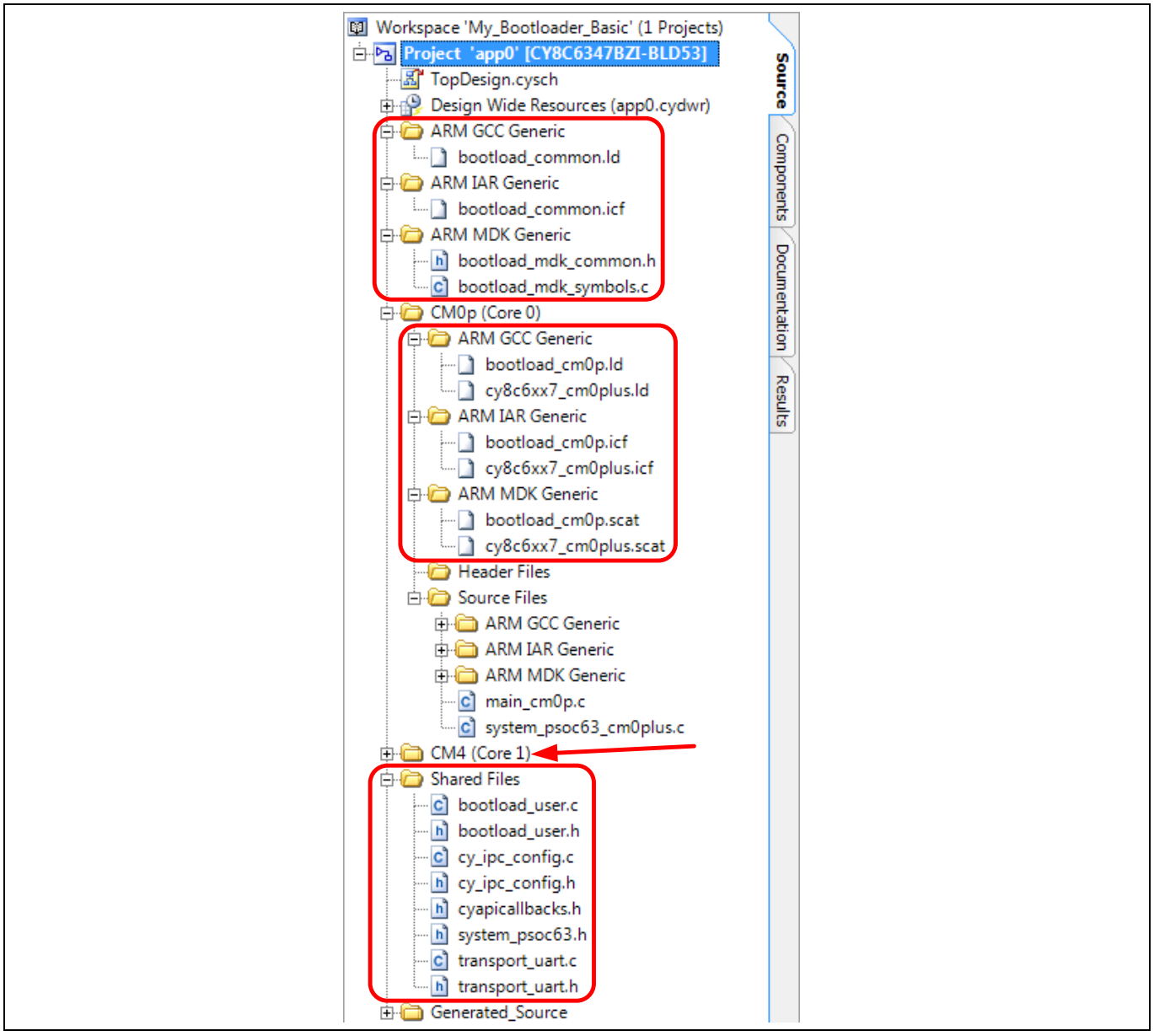


Figure 11　SDK ファイルを PSoC Creator プロジェクトに追加

6. インストールされたリンカー スクリプト ファイルはデフォルトでアプリケーション#0 (app0) のためにセットアップされます。その他のアプリケーションに対しては、アプリケーション番号を変更してファイルを編集します。以下の例は、GCC リンカーの app1 用に *bootload_cm0p.ld* と *bootload_cm4.ld* ファイルで行う編集を示します。

```
/*
* Bootloader SDK-specific: aliases regions, so the rest of the code does
not
* use application-specific memory region names
*/
REGION_ALIAS("flash_core0", flash_app1_core0);
REGION_ALIAS("flash",       flash_app1_core1);
REGION_ALIAS("ram",           ram_app1_core1);

/* Bootloader SDK-specific: sets App ID */
__cy_app_id = 1;
```

Keil µVision MDK リンカーはより複雑です。以下の例は、app1 用に *bootload_cm0p.scat* と *bootload_cm4.scat* ファイルで行う編集を示します。

```
; Flash
#define FLASH_START             CY_APP1_CORE0_FLASH_ADDR
#define FLASH_SIZE              CY_APP1_CORE0_FLASH_LENGTH

; Emulated EEPROM Flash area
#define EM_EEPROM_START         CY_APP1_CORE0_EM_EEPROM_ADDR
#define EM_EEPROM_SIZE          CY_APP1_CORE0_EM_EEPROM_LENGTH

; External memory
#define XIP_START               CY_APP1_CORE0_SMIF_ADDR
#define XIP_SIZE                CY_APP1_CORE0_SMIF_LENGTH

; RAM
#define RAM_START               CY_APP1_CORE0_RAM_ADDR
#define RAM_SIZE                CY_APP1_CORE0_RAM_LENGTH
```

以下は App1 用に *bootload_mdk_symbols.c* ファイルで行う編集です。

```
__cy_app_core1_start_addr   EQU __cpp(CY_APP1_CORE1_FLASH_ADDR)

/* Application number (ID) */
```

```
__cy_app_id                 EQU 1

/* CyMCUElfTool uses these to generate an application signature */
__cy_app_verify_start     EQU __cpp(CY_APP1_CORE0_FLASH_ADDR)
__cy_app_verify_length    EQU __cpp(CY_APP1_CORE0_FLASH_LENGTH +
                                    CY_APP1_CORE1_FLASH_LENGTH -
                                    __CY_BOOT_SIGNATURE_SIZE)
```

7. デフォルトのリンカー スクリプト ファイルの代わりにブートローダ リンカー スクリプト ファイルを使用するようプロジェクトのビルド設定を変更します。**Figure 12** に示すように、CM0+ CPU の *bootload_cm0p.ld* ファイルを選択します。ファイルへの相対パスを含めます。CM4 CPU の *bootload_cm4.ld* ファイルを選択します (図に示しません)。

Note: *リンカー スクリプト ファイルは GCC、Keil µVision MDK および IAR Embedded Workbench リンカー用に用意されています。この例は GCC リンカー用です。ご使用の IDE に適切なリンカー スクリプトを使用してください。*

Note: *このステップは、デバッグとリリースの両方のコンフィギュレーションに対して行う必要があります。*

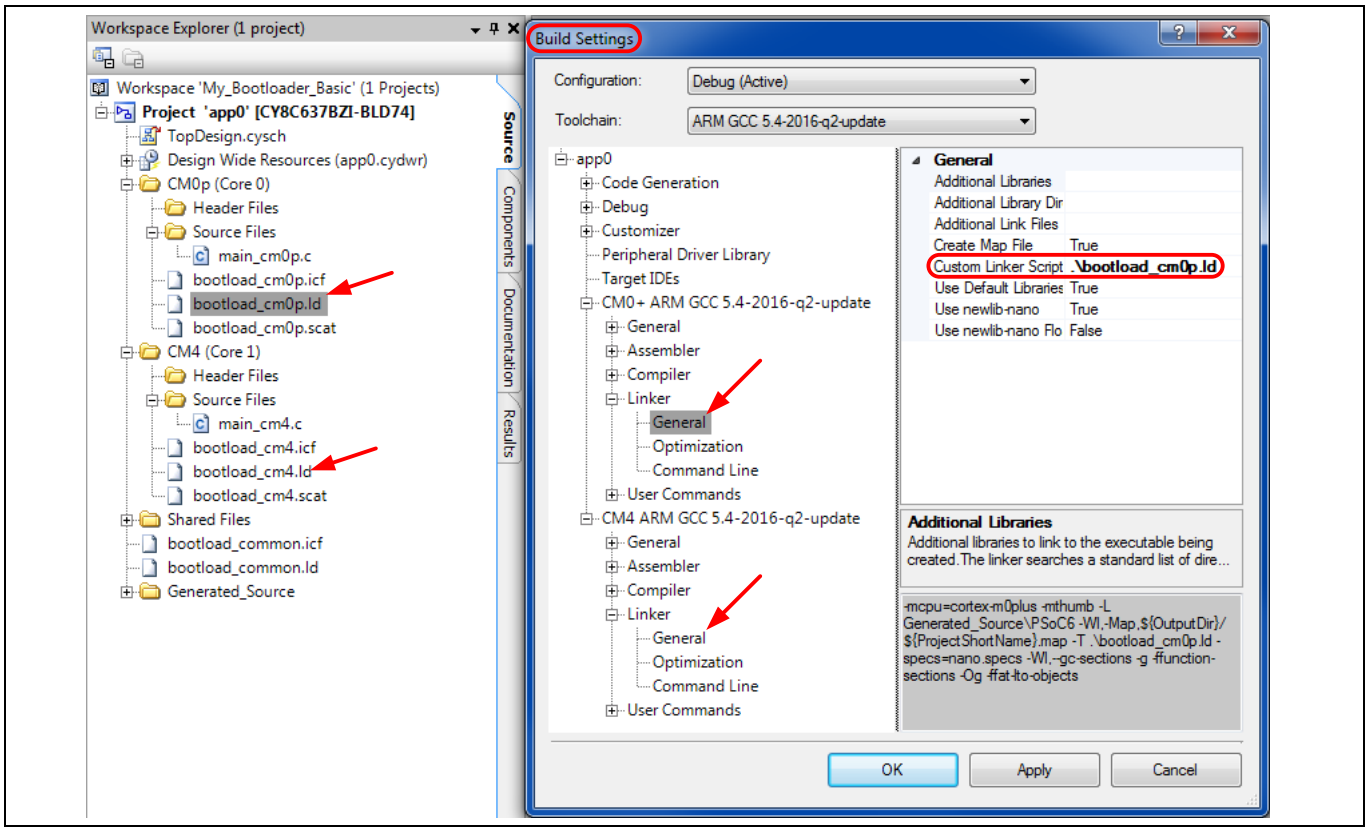


Figure 12 カスタムのリンカー スクリプトの PSoC Creator プロジェクト ビルド設定

8. ダウンロード可能アプリケーション プロジェクトの場合、サイプレス ユーティリティ プログラム *cymcuelftool.exe* を呼び出すためにビルド後のバッチ ファイルを追加します。**cymcuelftool** は PSoC Creator のインストールに含まれています。これは、ブートローダ ホストによってダウンロードされ

る *.cyacd2* ファイルを生成します (**Figure 5** を参照してください)。バッチ ファイルは CM4 バイナリにのみ適用されます。

Note: *バッチ ファイルの一例は、**CE213903**「PSoC 6 MCU Basic Bootloaders」などのブートローダ SDK サンプル コードを参照してください。通常、CE213903 \ Bootloader_Basic_App1.cydsn フォルダに保存されいます。また、バッチ ファイルの内容は **Application E** に記述されています。便宜のために、ファイルをコピーして PSoC Creator プロジェクト フォルダ app1.cydsn に貼り付けます。*

そして、**Figure 13** に示すように、PSoC Creator の Workspace Explorer ウィンドウで、バッチ ファイルをプロジェクトの *Shared Files* フォルダに追加し、また Cortex-M4 ビルドのビルド後のコマンドとして追加します。

Note: *このステップは、デバッグとリリースの両方のコンフィギュレーションに対して行わなければなりません。*

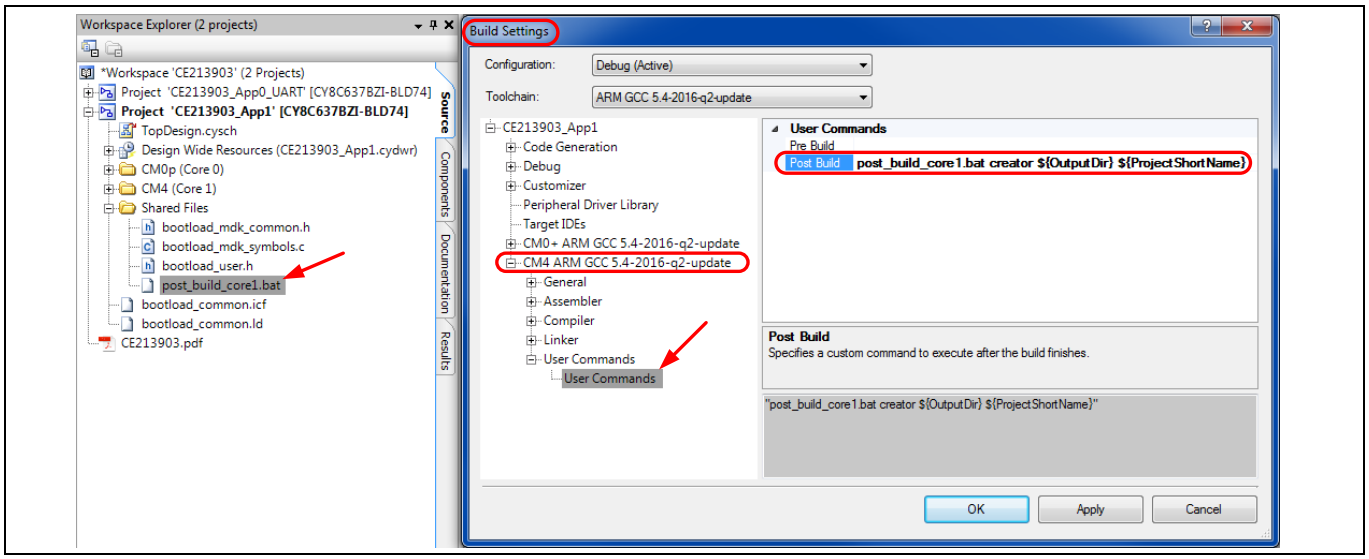


Figure 13 PSoC Creator ビルド後のコマンド

必要に応じて 2 つの *main_cmx.c* ファイルおよびその他のソース ファイルにコードを追加します。全体的な設計では、以下を検討してください。ダウンロード可能なアプリケーション プロジェクトの場合、*main_cm4.c* ファイルにグローバル配列を追加して、アプリケーションの署名を保存します。たとえば、最も単純で最も一般的な署名タイプであるチェックサムの場合、次を追加します。

```
/* This section holds signature data for application verification.
   For checksum verification, set the number of elements in the array to
1, and
   in bootload_common.ld set __cy_boot_signature_size = 4.
*/
CY_SECTION(".cy_app_signature") __USED static const uint32_t
                                cy_bootload_appSignature[1];
```

両方のメインファイルのコード全体で、次のことを考慮してください。

- 呼び出すブートローダ関数
  - `Cy_Bootload_DoBootload()`: 全体的なブートロード動作を行っている時にブロックします。ブートロード時にその他のタスクがない場合にこの関数を呼び出します。
  - `Cy_Bootload_Init()` の後に `Cy_Bootload_Continue()` への一連の呼び出し: `Cy_Bootload_Continue()` は、ホストからのコマンド パケットを受信／処理／応答している時にブロックします。ブートロード時に他のタスクがある場合にこれらの関数を呼び出します。
- 各アプリケーションが別のアプリケーションに制御を渡すかどうか。必要に応じてブートローダ SDK の API 関数 `ValidateApplication()` および `ExecuteApplication()` への呼び出しを追加します。

詳細については、PDL ブートローダ ドキュメントを参照してください。

Note: ***PSoC 6 MCU DFU サンプル コード**および**関連資料**に記載されているサンプル コードからコードをコピーアンドペーストできます。*

4.4 アプリケーションのビルドとプログラム

すべてのアプリケーションをビルドします。**Table 3** が ModusToolbox IDE に示すように、いくつかの出力ファイルが作成されます。 PSoC Creator の出力ファイルも同様です。 (**Figure 5** も参照)。すべての出力ファイルは、それぞれのプロジェクトのデバッグまたはリリースフォルダにあります。

Table 3 ModusToolbox IDE の DFU 出力ファイル (CE213903、PSoC 6 基本 DFU の例)

ファイル名	説明
App0, ModusToolbox IDE 用 DFU module 付き	
PSoC6DfuApp0xxx_mainapp_cm0p.elf	リンカー出力, CM0+ プロジェクト
PSoC6DfuApp0xxx_mainapp_cm0p_signed.elf	署名プロセス後 cymcuelftool 出力, CM0+ プロジェクト
PSoC6DfuApp0xxx_mainapp.elf	リンカー出力, CM4 プロジェクト
PSoC6DfuApp0xxx_mainapp_signed.elf	署名プロセス後 cymcuelftool 出力, CM4 プロジェクト
PSoC6DfuApp0xxx_mainapp_final.elf	cymcuelftool 最終出力, CM4 および CM0+出力の組み合わせこれは、ターゲットデバイスにプログラムされたファイルです。
App1, ModusToolbox IDE 用ダウンロード可能アプリケーション	
PSoC6DfuApp1_mainapp_cm0p.elf	リンカー出力, CM0+ プロジェクト
PSoC6DfuApp1_mainapp_cm0p_signed.elf	署名プロセス後 cymcuelftool 出力, CM0+ プロジェクト
PSoC6DfuApp1_mainapp.elf	リンカー出力, CM4 プロジェクト
PSoC6DfuApp1_mainapp_signed.elf	署名プロセス後 cymcuelftool 出力, CM4 プロジェクト
PSoC6DfuApp1_mainapp_final.elf	cymcuelftool 最終出力, CM4 および CM0+出力の組み合わせ
PSoC6DfuApp1_mainapp_dfu.elf	cymcuelftool 最終出力, CRC バイトで署名 *dfu_postbuild.bash* 参照
PSoC6DfuApp1_mainapp_dfu.cyacd2	cymcuelftool ダウンロード可能出力 *dfu_postbuild.bash* 参照 これは、デバイスファームウェア更新ホストツールへの入力です。

SDK の使用方法

ModusToolbox IDE または PSoC Creator を使用して、*App0..._mainapp_final.elf* ファイルをデバイスにプログラムします。デバイス プログラミングの詳細については、それらのツールのヘルプ メニューを参照してください。

その後、アプリケーション#0 の DFU モジュールをホスト プログラムと共に使用し、アプリケーションの*.cyacd2* ファイルをダウンロードしてデバイスにインストールできます。**Appendix A: DFU ホスト ツール**を参照してください。

5 PSoC 6 MCU DFU サンプル コード

本アプリケーション ノートに関連するサンプル コードがいくつかあります。それらは **DFU 操作**を実施する異なる方法を説明します。

サンプル コードと他のドキュメントの完全なリストおよび **www.cypress.com** 掲載のダウンロード リンクは、**関連資料**に記載されています。

Table 4 に、サンプル コードの概要リストを示します。

Table 4 PSoC 6 MCU DFU サンプル コードの一覧

CE #	タイトル	説明
ModusToolbox IDE および PSoC Creator 対応		
CE213903	PSoC 6 MCU の DFU Basic	基本的な DFU 操作を説明する一連の例。**Figure 6** のメモリ マップ A を参照してください。 UART、I^2C、SPI など様々な通信チャネルで、ホストからアプリケーションをダウンロード ダウンロードしたアプリケーションをユーザー フラッシュにインストール アプリケーションを検証してから、そのアプリケーションに制御を渡す
PSoC Creator のみ対応		
CE216767	PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity Bootloader	通信チャネルとして BLE を使用することを除き、CE213903 と同じ
CE220959	PSoC 6 MCU BLE Bootloader with External Memory	アプリケーションを外部メモリに一時的に保存してから、ユーザー フラッシュ内の最終保存場所にコピーすることを除き、CE216767 と同じ
CE220960	PSoC 6 MCU BLE Bootloader with Upgradeable Stack	BLE スタックがアップグレード可能であることを除き、CE216767 と同じ
CE221984	PSoC 6 MCU Dual-Application Bootloader	工場出荷時のデフォルト (「ゴールデン イメージ」) モードがある、2 つのアプリケーションのブートローディングを説明することを除き、CE213903 と同じ
CE222802	Bootloader with Encryption and Signing	検証のためにアプリケーションが暗号化され、シグネチャ付けられることを除き、CE213903 と同じ。ブートローダはアプリケーションを復号し、そのシグネチャを検証
以下のサンプル コードは計画中または開発中		
CE2xxxxx	USB HID DFU	通信チャネルとして USB HID を使用することを除き、CE231903 と同じ
CE2xxxxx	USB Mass Storage DFU	通信チャネルとして USB 大容量ストレージを使用することを除き、CE231903 と同じ
CE2xxxxx	DFU with Multiple Applications	各アプリケーションは別のアプリケーションを更新できることを除き、CE221984 と同じ。**Figure 6** のメモリマップ B を参照。
CE2xxxxx	PSoC 6 MCU DFU with External Memory Source	DFU ソースが外部メモリであることを除き、CE213903 と同じ

CE #	タイトル	説明
CE2xxxxx	Embedded Host Program	「**AN60317**, PSoC 3 and PSoC 5LP I²C Bootloader」で説明するものと同様

サンプル コードのほとんどは、「App0」と「App1」と呼ばれる複数の個別のアプリケーションを含みます。「App2」、「App3」などの追加のアプリケーションが含まれる場合があります。各アプリケーションは ModusToolbox IDE の個別のアプリケーションまたは PSoC Creator の個別のプロジェクトです。

一般的には、すべてのアプリケーションは同じ ModusToolbox または PSoC Creator ワークスペースにあります。**アプリケーションを設計**で説明したように、プロジェクトは個別のワークスペースまたはコンピュータの個別の場所にある場合があります。

通常、App0 は DFU 操作を行い、別のアプリケーションをダウンロードしてインストールします。注目すべき例外の 1 つは、アップグレード可能なスタックを備えた BLE ブートローダ **CE220960** です。このサンプル コードでは、App1 は App2 をダウンロードしてインストールします。App0 は制御を他のアプリケーションの 1 つに転送するだけです。

基本サンプル コード **CE213903** には、複数の「_App0_ … 」プロジェクトがあり、それぞれのプロジェクトは異なる通信チャネルを持っています。BLE や USB などの高度な通信チャネルは他のサンプル コードで説明されます。

サンプル コードは **CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit** に対応しています。通常、どのアプリケーションが実行中であるかを示すために、アプリケーションは異なる速度または異なる色でキットの LED を点滅させます。サンプル コードを別のキットまたはシステムに移植するのは簡単な作業です。

アプリケーションは一般的に、キット ボタンを 0.5 秒押すことで 1 つのアプリケーションから別のアプリケーションに制御を渡せるように設計されています。BLE ブートローダ サンプル コードでは、即時アラート サービス (IAS) を使用してアプリケーション間の制御権移転を行います。

5.1 サンプル コードのビルド方法

以下は、**Table 4** にリストアップした各 DFU サンプル コードをビルドする手順です。これらの手順は、アプリケーションごとに任意に適合させられます。詳細については、**SDK の使用方法**を参照してください。

Note: *サンプル コードをビルドする場合、多くの場合、既存のサンプル コードの一部をプロジェクトにコピーするのが最も簡単です。コピーされた部分は、アプリケーションに合わせて変更できます。コピーされる部分は、各命令セットにリストされています。*

手順はサイプレスの ModusToolbox および PSoC Creator IDE をベースとしていますが、その他の IDE に適合させられます。

5.2 PSoC 6 MCU の基本 DFU

本節では、サンプル コード **CE213903** の基本 DFU をビルドする方法を示します。5 つの一般的なステップがあります。

1. App0 およびワークスペースを作成
2. App0 を DFU に設定
3. App1 を追加
4. App1 をインストール可能アプリケーションに設定
5. DFU とアプリケーションをビルドおよびテスト

これらの手順は、ModusToolbox IDE と PSoC Creator の両方に適用されます。

5.2.1 ModusToolbox の手順

1. **App0 およびワークスペースを作成**

ModusToolbox IDE を起動し、**Figure 14** に示すように、アプリケーションのワークスペースを選択または作成します。

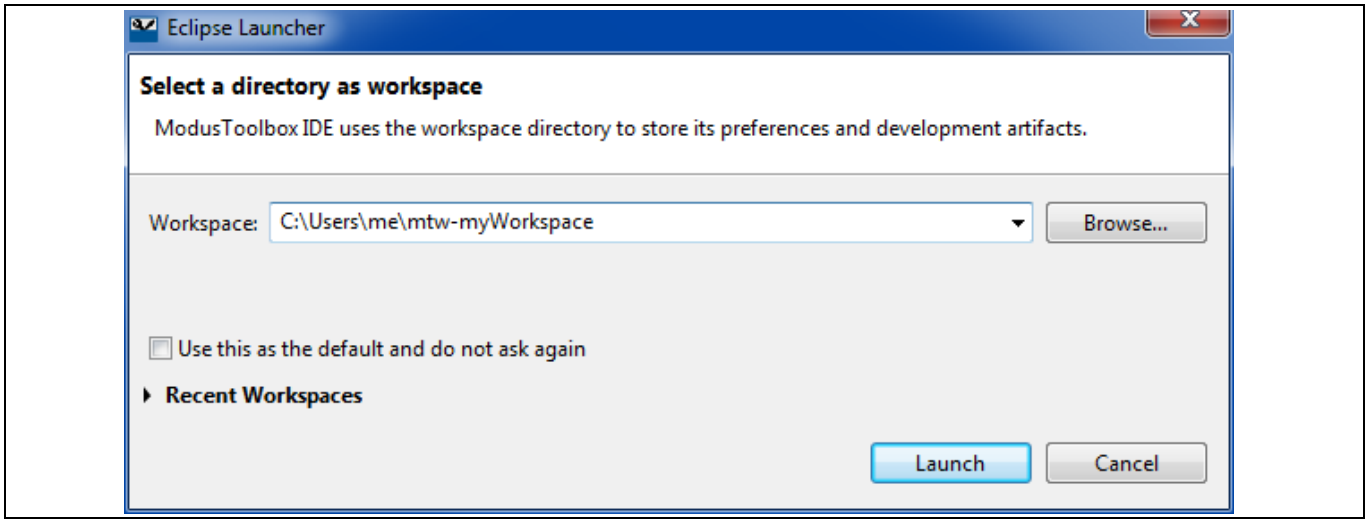


Figure 14 ModusToolbox ワークスペースを定義

新しいアプリケーションを作成します。これを行う最も簡単な方法は、**Quick Panel** ウィンドウで **New Application** をクリックすることです。次のダイアログで、**Dev/Eval Kit** をクリックし、**CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit** のために **CY8CKIT-062-BLE** を選択します。 **Next** をクリックします。

次のダイアログ **Starter Application** で、**EmptyPSoC6App** を選択し、**Name** を変更します。CE213903 は、「PSoC6DfuApp0Uart」、「PSoC6DfuApp0I2c」、および「PSoC6DfuApp0Spi」を使用します。これが DFU を実行するアプリケーションであることを示す名前を選択します。ホスト通信チャネルも示せます。**Next** をクリックします。Summery ダイアログで選択内容を確認し、**Finish** をクリックします。

2. **App0 を DFU に設定**

アプリケーションに DFU 機能を追加するために、以下を行ってください。

- 設計にホスト通信チャネルを追加
- 必要に応じて、他の周辺機器を追加
- DFU コードを追加

2a. 通信チャネルを追加: プロジェクト<name>_config **の** *design.modus* を開く。チャネルタイプを選択し、CY8CKIT-062-BLE キットで動作するように設定します。

- UART の場合： デフォルトから次の変更を加えて UART を SCB 5 に追加します。Oversample = 12、Clock =任意の便利なクロック、RX = P5[0]、TX = P5[1]。オプションで、UART および UART 端子に名前を割り当てられます。
- I^2C の場合： デフォルトから次の変更を加えて I2C を SCB 3 に追加します。Data Rate (kbps) = 400、Clock =任意の便利なクロック、SCL = P6[0]、SDA = P6[1]。オプションで、I2C および I2C 端子に名前を割り当てられます。
- SPI の場合： デフォルトから次の変更を加えて SPI を SCB 6 に追加します。Clock = 任意の便利なクロック、SCLK = P12[2]、MOSI = P12[0]、MISO = P12[1]、SS1 = P12[4]。オプションで、SPI および SPI 端子に名前を割り当てられます。

2b. 必要に応じて他の周辺機器を追加: CE213903 との互換性のために、キット LED およびボタンサポート用の端子を構成します。

ピン P0[3]を次のように構成します。name= KIT_RGB_R、Drive Mode = Strong Drive Input Buffer off。Initial Drive State = High (1) であることを確認します。

ピン P0[4]を次のように構成します。name = KIT_BTN1、Drive Mode = Resistive Pull-Up Input buffer on。Initial Drive State = High (1) であることを確認します。

2c. DFU コードを追加: プロジェクト<name>_mainapp を右クリックし、**ModusToolbox Middleware Selector** を選択します。ミドルウェア **DFU SDK Core**、および **DFU SDK transport UART** または他のトランスポートオプションのいずれかを選択します。

<name>_mainapp_cm0p についても同じことを行い、**DFU SDK Core** のみを選択します。必要なソース *.c*、*.h*、およびリンカースクリプト (例: *.ld*) ファイルは、両方のプロジェクトに自動的に追加されます。

I²C および SPI チャネルの場合、次のように、<name>_mainapp プロジェクトの *Source* フォルダでファイル *dfu_user.c* を編集します。

- `#include "transport_uart.h"` を次に変更
  `#include "transport_i2c.h"` または
  `#include "transport_spi.h"`
- “`UART_Uart`” の 5 つのインスタンスを “`I2C_I2c`” または “`SPI_Spi`” に変更

DFU コードを<name>_mainapp および<name>_mainapp_cm0p の *main.c* ファイルに統合します。これを行う最も簡単な方法は、サンプル コード **CE213903** からコードをコピーして貼り付け、アプリケーションのコードを変更することです。

プロジェクト<name>_mainapp を右クリックして、**Proparties** を選択します。C/C++ **Build** > **Settings** > **Tool Settings** > **GNU ARM Cross C Linker** > **General** で、**Script files** エントリのファイルを *dfu_cm4.ld* に変更し、プロジェクトのカスタムスクリプトを使用してリンクします。*dfu_cm0p.ld* のためプロジェクト <name>_mainapp_cm0p についても同じことを行います。

2d. プロジェクトをビルド: これを行う最も簡単な方法は、<name>_mainapp を選択し、Quick Panel で **Build <name> Application** をクリックすることです。

3. App1 を追加

前述したように、ModusToolbox IDE では、、DFU アプリケーションと同じワークスペース、または別のワークスペース、フォルダ、またはその両方にアプリケーションを追加できます。下記の手順は、DFU App0 と同じワークスペースに App1 を追加する方法を示します。

Note: *同じ DFU に対応しているインストール可能アプリケーションはいくつでも作成できます。アプリケーションの開発を始める前に、システム開発全体のニーズに合わせて DFU とアプリケーションの計画作成をお勧めします。**システムでアプリケーションを決定**を参照してください。*

前述の方法と同じ方法で新しいアプリケーションを作成します。App0 の場合と同じキットを選択します。名前を変更します。CE213903 は「PSoC6DfuApp1」を使用します。これがダウンロード可能なアプリケーションであることを示す名前を選択します。

4. App1 をインストール可能アプリケーションに設定

4a. 必要に応じて他の周辺機器を追加: このアプリケーションの *design.modus* を編集します。CE213903 との互換性のために、キット LED およびボタンサポート用の端子を構成します。

ピン P0[3]を次のように構成します。name = KIT_RGB_R、Drive Mode = Strong Drive Input Buffer off。Initial DriveState = High (1)であることを確認します。

ピン P0[4]を次のように構成します。name = KIT_BTN1、Drive Mode = Resistive Pull-Up Input buffer on。Initial Drive State = High (1)であることを確認します。

4b. DFU コードを追加: プロジェクト<name>_mainapp を右クリックし、**ModusToolbox Middleware Selector** を選択します。ミドルウェア **DFU SDK Core** を選択します。<name>_mainapp_cm0p についても同じことを行います。 必要なソース.*c*、.*h*、およびリンカースクリプト (例: .*ld*) ファイルは、両方のプロジェクトに自動的に追加されます。DFU SDK Core のアプリケーションへの追加は、App0 への制御の転送を有効にするためにのみ行われます。

プロジェクト<name>_mainapp の *dfu_cm4.ld* ファイルとプロジェクト<name>_mainapp_cm0p の *dfu_cm0p.ld* ファイルを次のように編集します。これにより、App1 のアプリケーションが変更され、App1 のフラッシュメモリと RAM メモリが適切に配置されます。

```
/*
* DFU SDK specific: aliases regions, so the rest of code does not use
* application specific memory region names
*/
REGION_ALIAS("flash_core0", flash_app1_core0);
REGION_ALIAS("flash",       flash_app1_core1);
REGION_ALIAS("ram",           ram_app1_core1);


/* DFU SDK specific: sets an app Id */
__cy_app_id = 1;
```

Note: *これらの.ld ファイルで他のものを変更しないことが重要です。上記の編集を除き、それらは App0 .ld ファイルと同じでなければなりません。*

DFU コードを<name>_mainapp および<name>_mainapp_cm0p の *main.c* ファイルに統合します。これを行う最も簡単な方法は、サンプル コード CE213903 からコードをコピーして貼り付け、アプリケーションのコードを変更することです。

サンプル コードのファイル *dfu_postbuild.bash* をプロジェクト<name>_mainapp にコピーします。**付録 E. ビルド後のファイル一覧**を参照してください。

プロジェクト<name>_mainapp_cm0p を右クリックして、**Proparties** を選択します。**C/C++ Build** > **Settings** > **Tool Settings** > **GNU ARM Cross C Linker** > **General** で、**Script files** エントリのファイルを *dfu_cm0p.ld* に変更し、プロジェクトのカスタム スクリプトを使用してリンクします。

dfu_cm4.ld のためプロジェクト<name>_mainapp についても同じことを行います。次に、同じウィンドウで **Build Steps** を選択し、**Post-build steps Command** を次のコマンドに置き換えます。

```
${cy_sdk_install_dir}/tools/modus-shell-1.0/bin/bash --norc --noprofile
${workspace_loc:/PSoC6DfuApp1_mainapp}/dfu_postbuild.bash
${cy_sdk_install_dir}/tools/cymcuelftool-1.0/bin/cymcuelftool
${workspace_loc:/PSoC6DfuApp1_mainapp_cm0p}/${config_name:PSoC6DfuApp1_ma
inapp_cm0p}/PSoC6DfuApp1_mainapp_cm0p.elf
${workspace_loc:/PSoC6DfuApp1_mainapp}/${config_name:PSoC6DfuApp1_mainapp
}/PSoC6DfuApp1_mainapp.elf ARM_CM4
```

出力*.cyacd2* ファイルを作成します。これは、DFU 用のアプリケーションをダウンロードするために使用されます。**付録 C. .cyacd2 ファイル フォーマット**を参照してください。

4c. プロジェクトをビルド: これを行う最も簡単な方法は、<name>_mainapp を選択し、Quick Panel で **Build <name> Application** をクリックすることです。

5. **DFU とアプリケーションをテスト**

App0 を **CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit** にプログラムします。次に、Device Firmware Update Host Tool (DHT)を使用して App1 をキットにダウンロードします。**付録 A. DFU ホスト ツール**のツールの使用手順を参照してください。

CE213903 の Operation セクションの手順を使用して、DFU およびアプリケーションをテストできます。

5.2.2 PSoC Creator の手順

1. **App0 およびワークスペースを作成**

PSoC Creator プロジェクトとそれを含むワークスペースを同時に作成できます。PSoC Creator で、**File** > **New** > **Project...**を選択します。**Figure 15** に示すように、Select project type パネルとともに Create Project ウィンドウが表示されます。

A. **Target device** をクリックします。

B. 最初のドロップダウン メニューで、**PSoC 6** を選択します。

C. 次のドロップダウン メニューで、**CY8C6347BZI-BLD53** を選択します。これは **CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit** にインストールされた PSoC 6 MCU デバイスです。

D. **Next** をクリックします。

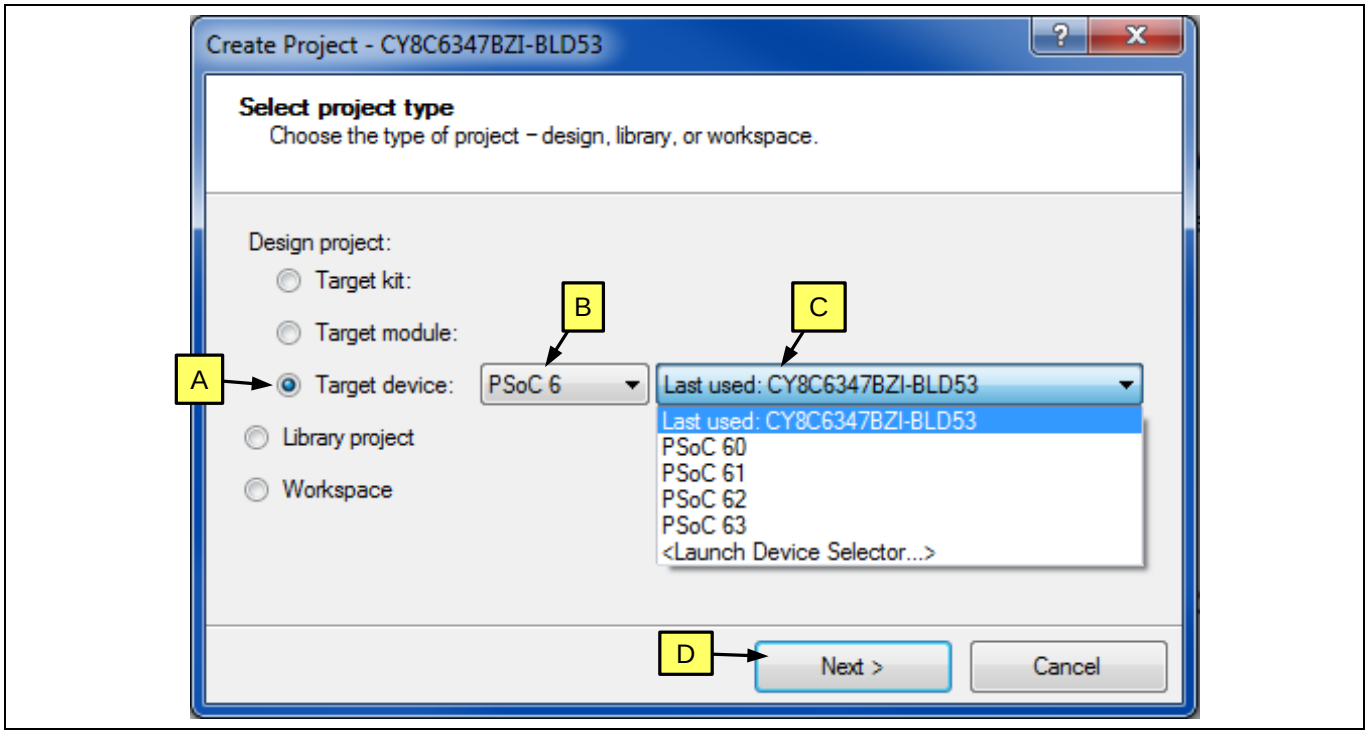


Figure 15 PSoC Creator のターゲット デバイスを選択

Select project template と **Set target IDE(s)** の次の 2 つのパネルで、**Next** をクリックします。**Figure 16** に示すように、Create Project パネルが表示されます。

Workspace name に入力し、**Location** を選択します。**Project name** を「app0」にセットします。**Finish** をクリックします。

指定した場所に、新しいフォルダが作成されます。フォルダ名はワークスペース名と同じです。ワークスペース フォルダ内に *app0.cydsn* フォルダが作成されます。*.cydsn* フォルダには、すべてのプロジェクト ファイルが含まれます。

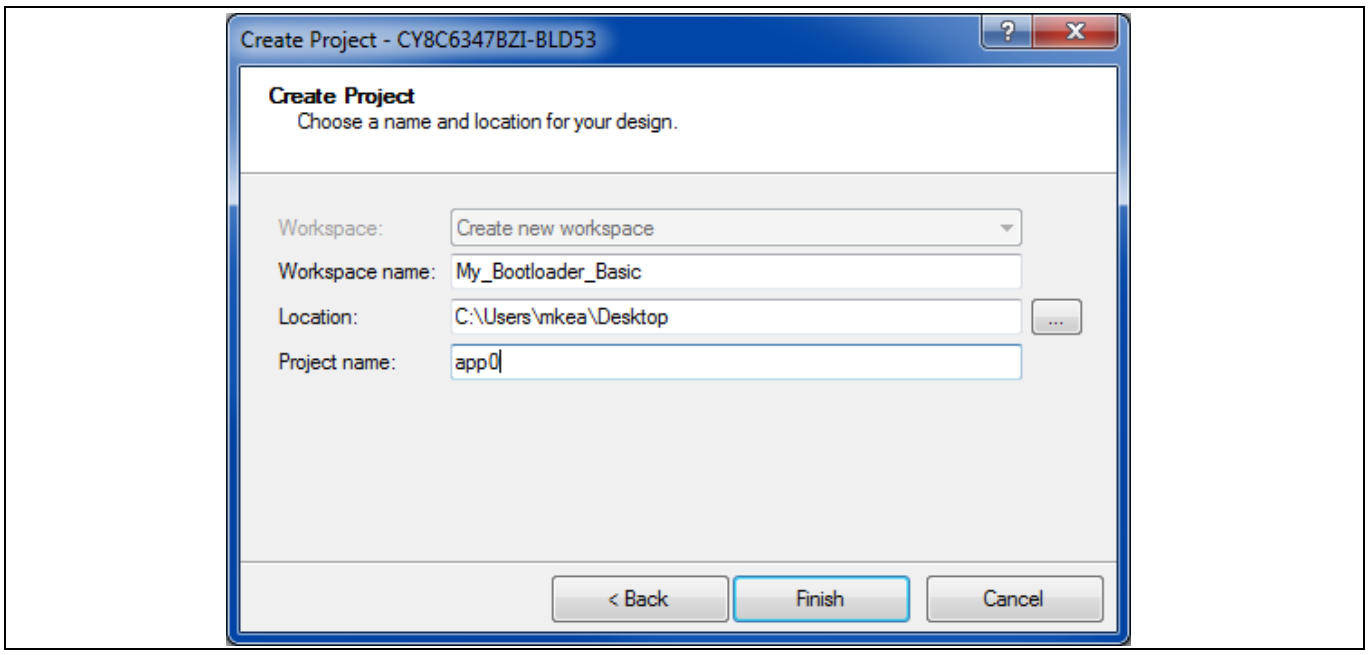


Figure 16 PSoC Creator プロジェクトを作成

新規作成のワークスペースとプロジェクト ファイルは Workspace Explorer ウィンドウに示されます。**Figure 11** を参照してください。

2. **App0 をブートローダに設定**

アプリケーションにブートローダ機能を追加するために、以下を行ってください。

- プロジェクト回路図に通信コンポーネントを追加。このコンポーネントをブートローダ ホストとの通信に使用
- アプリケーションの必要に応じて、プロジェクト回路図に他のコンポーネントを追加
- プロジェクトにブートローダ SDK とテンプレート ファイルを追加

2a. 通信コンポーネントを追加: プロジェクト回路図を開く、または Workspace Explorer ウィンドウ内の *TopDesign.cysch* ファイルをダブルクリックします。

Component Catalog を開き、**Figure 17** に示すように、UART などの所望の通信コンポーネントを探します。 そのコンポーネントを回路図にドラッグします。

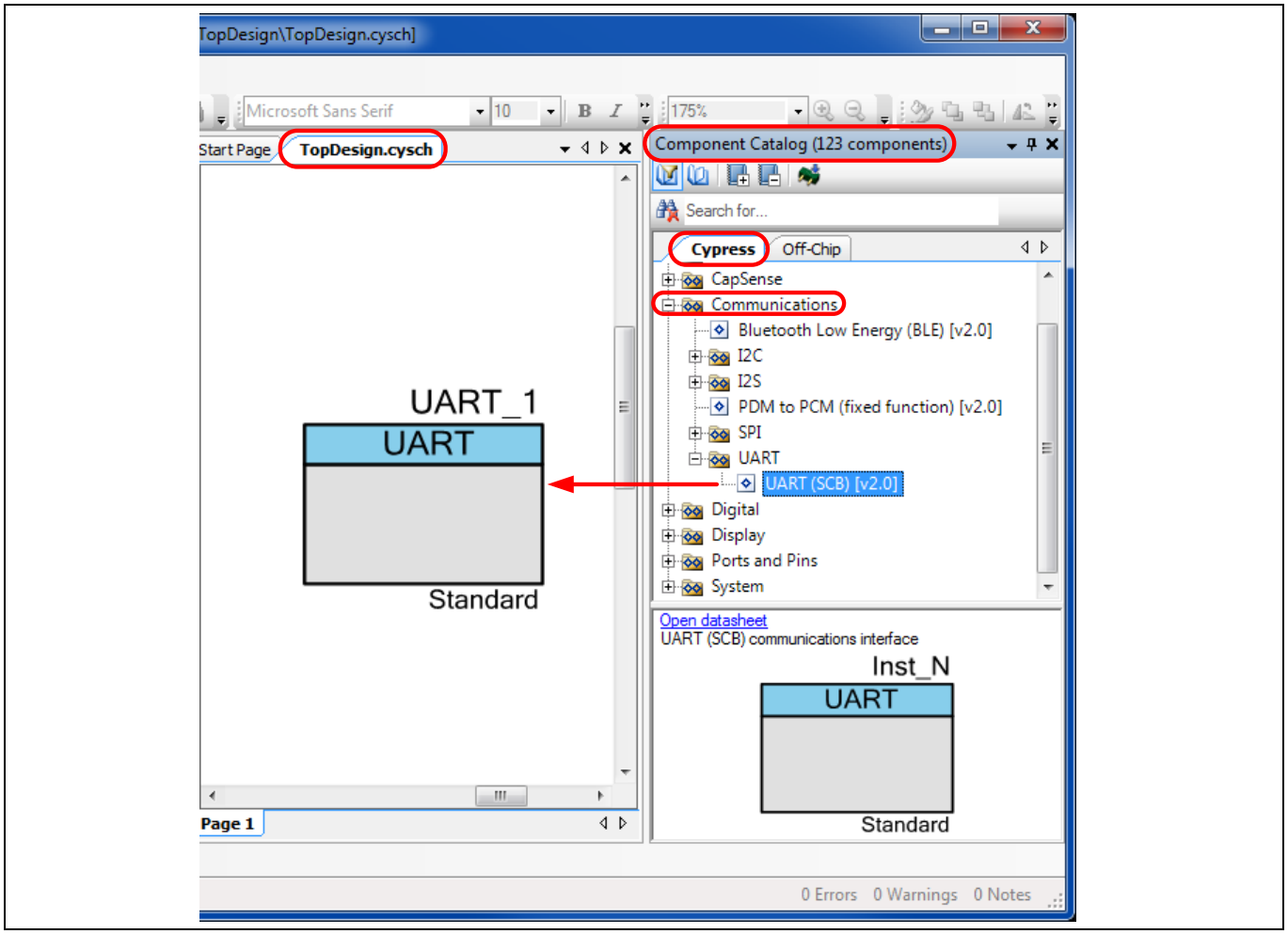


Figure 17　ブートローダ プロジェクトに PSoC Creator 通信コンポーネントを追加

Figure 18 に示すように、回路図上のコンポーネントをダブルクリックし、ボーレートやビット数などのパラメーターを設定します。

Figure 18 は、本サンプル コードの変更されたパラメーター設定を示します。**Name** が UART_1 から UART に、I2C_1 から I2C に、や SPI_1 から SPI に変更されたことに注意してください。これは、後でプロジェクトにコピーするデフォルトのブートローダ SDK ファイルを使用するために推奨されています。

コンポーネントの設定を完了したら、**OK** をクリックします。

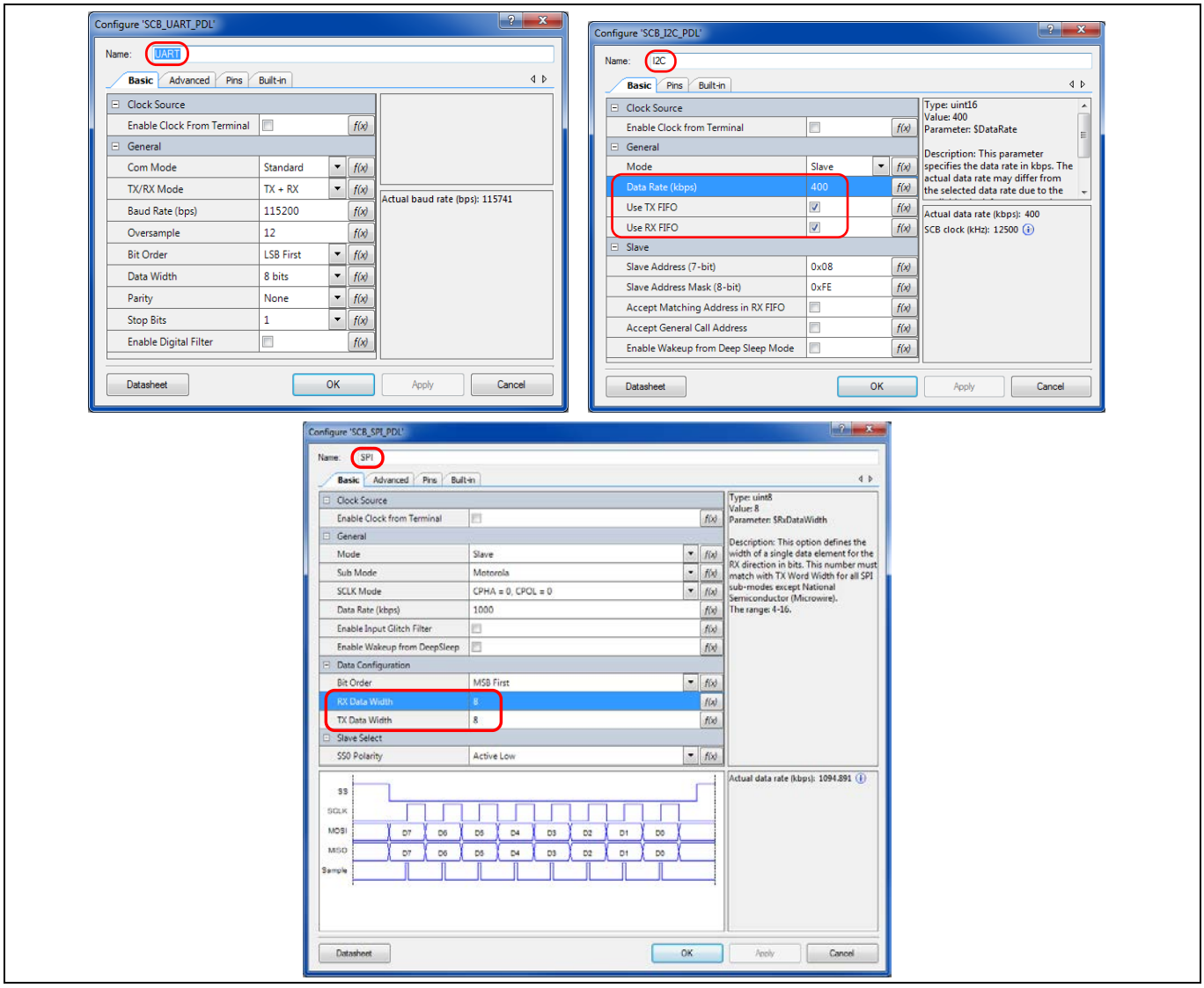


Figure 18　PSoC Creator 通信コンポーネントの設定: UART、I2C、または SPI

2b. 他のコンポーネントを追加: プロジェクト回路図に LED やボタン ピンなどの他の PSoC Creator コンポーネントを追加します。そのための最も簡単な方法は、サンプル コード **CE213903** からプロジェクト回路図の部分をコピーアンドペーストしてから、アプリケーションの必要に応じて回路図を修正することです。

Design Wide Resources ウィンドウの **Pins** タブで、UART、I2C、SPI およびその他のコンポーネント端子を適切な物理端子に接続します。手順については、**CE213903** およびキットのガイドを利用してしてください。

2c. プロジェクトにブートローダ コードを追加: まず、プロジェクトにブートローダ SDK を組み込みます。**Figure 10** を参照してください。

次に、**Build** > **Generate Application** を選択します。完了すると、プロジェクトは **Figure 11** のようになりますが、いくつかのスタートアップおよびその他の非ブートローダ ファイルが追加されました。

I²C および SPI ブートローダの場合、プロジェクトの *Shared Files* フォルダ内の *bootload_user.c* ファイルを以下のように編集してください。

- `#include "transport_uart.h"` を `#include "transport_i2c.h"` または `#include "transport_spi.h"` に編集
- “`UART_Uart`” の 5 つのインスタンスを “`I2C_I2c`” または “`SPI_Spi`” に変更

main_cm0p.c および *main_cm4.c* ファイルにブートローダ コードを追加します。そのための最も簡単な方法は、サンプル コード **CE213903** からコードをコピーアンドペーストしてから、アプリケーションに応じてコードを修正することです。

2d. プロジェクトをビルド: テンプレート リンカー スクリプト ファイルを使用するようプロジェクト ビルド設定を変更します。**Figure 12** を参照してください。次に、**Build** > **Build app0** を選択します。

3. **App1 を追加**

前述したように、PSoC Creator では、他のプロジェクト (すなわち、アプリケーション) を、ブートローダ アプリケーションと同じワークスペースまたは別のワークスペース、フォルダまたは両方に追加できます。下記の手順は、ブートローダ App0 と同じワークスペースに App1 を追加する方法を示します。

Note: *同じブートローダに対応しているインストール可能アプリケーションはいくつでも作成できます。アプリケーションの開発を始める前に、システム開発全体のニーズのためにブートローダおよびアプリケーション計画の開発をお勧めします。**システムでアプリケーションを決定**を参照してください。*

PSoC Creator の Workspace Explorer ウィンドウで、**Workspace** を右クリックして、**Figure 19** に示すように **Add** > **New Project...**を選択します。

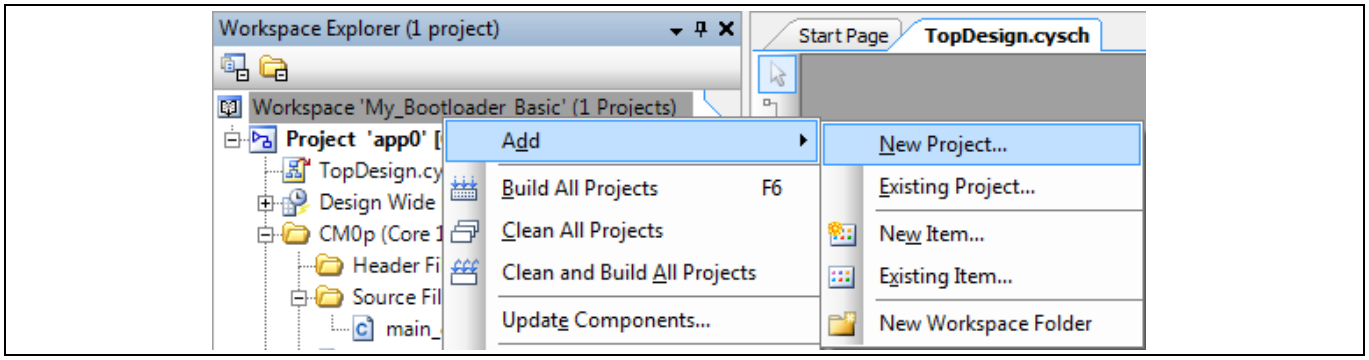


Figure 19 PSoC Creator ワークスペースに新規プロジェクトを追加

Select project type パネルとともに Create Project ウィンドウが表示されます。**Figure 15** を参照してください。選択するデバイスが、App0 と同じようにしてください。**Next** をクリックします。

Select project template と **Set target IDE(s)** の次の 2 つのパネルで、**Next** をクリックします。Create Project パネルが表示されます。

Figure 20 に示すように、**Project name** を「app1」にセットします。デフォルトの Location はワークスペース フォルダです。**Finish** をクリックします。

ワークスペース フォルダ内に新しい *app1.cydsn* フォルダが作成されます。*.cydsn* フォルダには、すべてのプロジェクト ファイルが含まれます。

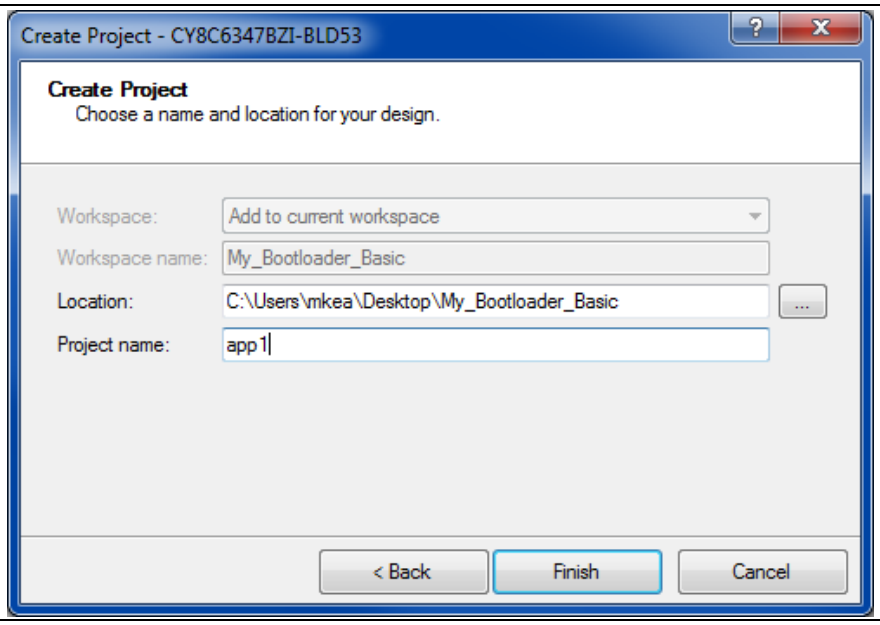


Figure 20　2 番目の PSoC Creator プロジェクトを作成

4. **App1 をインストール可能アプリケーションに設定**

Note: *App1 は、App0 と同じツールチェーン(GCC または MDK) でビルドする必要があります。そうしないと、アプリケーションの制御権移転が失敗することがあります。*

4a. プロジェクト回路図に LED やボタン ピンなどの他のコンポーネントを追加: そのための最も簡単な方法は、サンプル コード **CE213903** からプロジェクト回路図の部分をコピーアンドペーストしてから、アプリケーションに応じて回路図を修正することです。

Design Wide Resources ウィンドウの **Pins** タブで、コンポーネント ピンを適切な物理ピンに接続します。手順については、CE213903 およびキットのガイドを利用してしてください。

4b. プロジェクトにブートローダ SDK とテンプレート ファイルを追加: プロジェクトにブートローダ SDK を組み込みます。**Figure 10** を参照してください。**Core** ボックスのみをオンにします。App1 がダウンロード可能アプリケーションであり、ブートローダ機能を持っていないからです。

次に、**Build** > **Generate Application** を選択します。完了すると、プロジェクトは **Figure 11** のようになりますが、いくつかのスタートアップおよびその他の非ブートローダ ファイルが追加されました。テンプレート リンカー スクリプト ファイルを使用するようプロジェクト ビルド設定を変更します。**Figure 8** を参照してください。

4.3.2 節のステップ 8 で説明したように、*Shared Files* フォルダにビルド後のバッチ ファイルを追加します。その後、Cortex-M4 ビルドにビルド後のコマンドを追加します。**Figure 13** を参照してください。

4c. ファイルを編集: **4.3 節のステップ 6** で説明したようにリンカー スクリプト ファイルを編集します。

main_cm0p.c および *main_cm4.c* ファイルにブートローダ コードを追加します。そのための最も簡単な方法は、サンプル コード **CE213903** からコードをコピーアンドペーストしてから、アプリケーションに応じてコードを修正することです。

4.3.2 節のステップ 8 で説明されているように、アプリケーション署名用の *main_cm4.c* ファイルに配列が追加されていることを確認します。

4d. プロジェクトをビルド: すべてのビルド設定の変更およびファイルの編集が完了した後、**Build** > **Build app1** を選択します。

5. **ブートローダとアプリケーションをテスト**

CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit に app0 をプログラムします。次に、ブートローダ ホスト プログラム (BHP) を使用して app1 をキットにブートロードします。BHP の使用方法については、**付録 A. DFU ホスト ツール**を参照してください。

ブートローダおよびアプリケーションのテスト方法については、**CE213903** ドキュメントの Operation セクションを参照してください。

5.3 PSoC 6 MCU BLE のブートローダ

本節では、サンプル コードで 3 つの異なる BLE ブートローダをビルドする方法を示します。

- **CE216767**, PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity Bootloader
- **CE220959**, PSoC 6 MCU BLE Bootloader with External Memory
- **CE220960**, PSoC 6 MCU BLE Bootloader with Upgradeable Stack (**5.3.1** 参照)

これらのサンプル コードは **基本ブートローダのサンプル コード**に似ていますが、以下の機能が追加されます。

- CE216767 は **Figure 21** に示すように、新しいアプリケーションをフラッシュに直接ダウンロードします。
- CE220959 は **Figure 22** に示すように、新しいアプリケーションを外部 NVM (キット IC U4、サイプレス 512M ビット シリアル NOR フラッシュ) 内の一時的なストレージにダウンロードします。その後、PSoC 6 MCU デバイス フラッシュ内の最終保存場所にコピーします。
- **Figure 21** と **Figure 22** に示すように、ブートローダ (App0) とアプリケーション (App1) それぞれはそれ自身の BLE スタック コードのコピーを持っています。各コピーはフラッシュの 128KB 以上を占めます。単一の共有 BLE スタックを持つブートローダについては、**アップグレードスタック付き BLE ブートローダ**を参照してください。

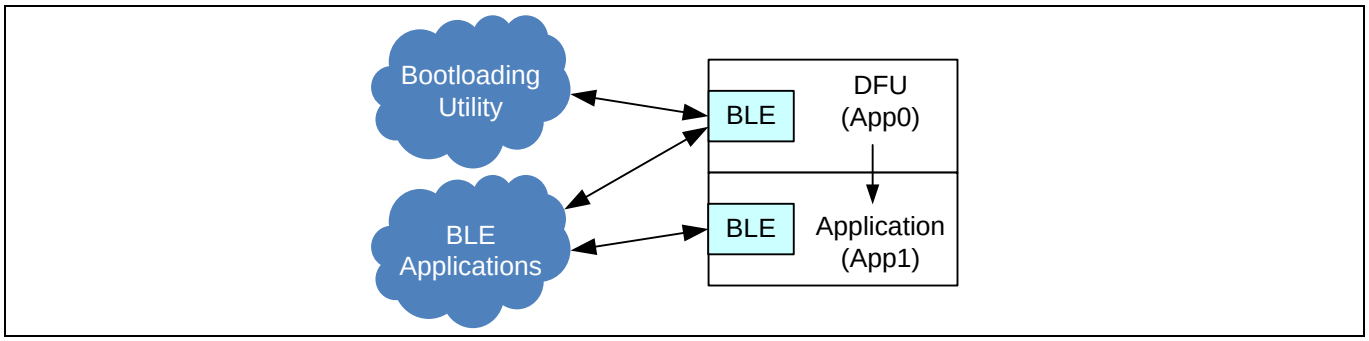


Figure 21 CE216767 のデータと制御フロー

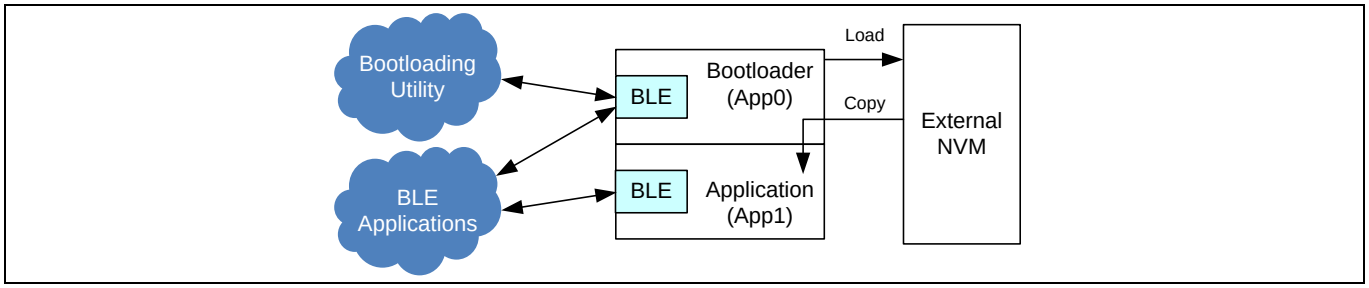


Figure 22 CE220959 のデータと制御フロー

- BLE スタック コードは 1 つの CPU または両方の CPU によって実行されます。これは、BLE コンポーネント コンフィギュレーション ダイアログ内の選択可能なオプションです。サンプル コードはこの機能を説明します。App0 で、CM4 CPU は BLE スタックを実行し、App1 で、CM0+ CPU は BLE スタックを実行します。
- App0 および App1 はそれらの BLE コンポーネント コンフィギュレーションで異なるサービスを有効にします。
  - App0: ブートローダおよび即時アラート サービス (IAS)。IAS サービスは、アプリケーションがブートロード中でない時に実装されます。
  - App1: ヒューマン インターフェース デバイス (HID) および即時アラート サービス (IAS)

  両方のアプリケーションは IAS を使用してアプリケーションからアプリケーションへ制御権を渡します。
- サンプル コードは **CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit** リソースを広く利用します。すべての 3 つの RGB LED、ユーザー ボタン SW2、BLE サブシステムおよび USB-UART ブリッジは両方のアプリケーション (App0 と App1) によって使用されます。

サンプル コードをビルドするために、以下を行ってください。手順の詳細については、**PSoC 6 MCU の基本ブートローダ**または **SDK の使用方法**を参照してください。

1. **Figure 23** に示すように、App0 およびワークスペースを作成します。

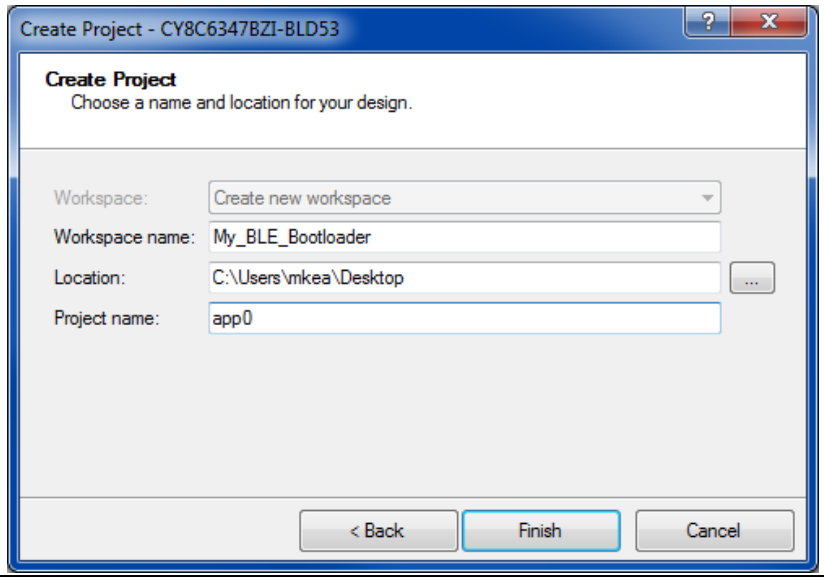


Figure 23 PSoC Creator の BLE ブートローダ App0 作成

2. App0 Top Design 回路図に BLE コンポーネントを追加してから、いずれかのサンプル コードの BLE コンポーネント セクションに応じてコンポーネントを設定します。特に、デフォルト コンフィギュレーションから以下の変更を行ってください。

- (オプションであるが、デフォルトのブートローダ SDK ファイルに対応するために推奨される) コンポーネント名を「BLE」に変更します。
- General タブ (**Figure 24**): CPU core: Single core (Complete Component on CM4)

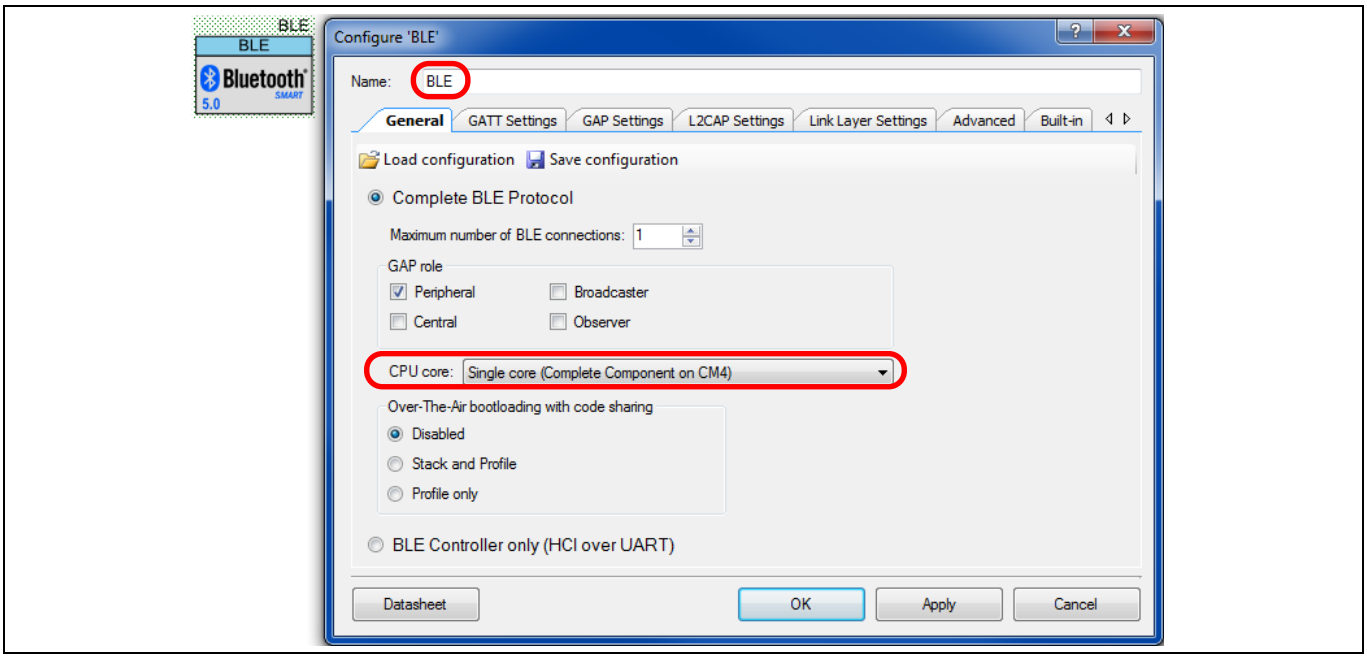


Figure 24　General タブでの PSoC Creator BLE コンポーネントの設定

- **GATT Settings** タブ (**Figure 25**):
  - Generic Access、Peripheral Preferred Connection Parameters:
    - Minimum Connection Interval: **0x000C**
    - Maximum Connection Interval: **0x000C**
    - Connection Supervision Timeout Multiplier: **0x00C8**

    これらの間隔は、ブートローディング時間を最小限に抑えるように選択します。アプリケーションの必要に応じて適応します。
  - GATT ツリーの **Server** サーバーを右クリックし、ブートローダ サービスを追加します。オプションで、即時アラート サービスを追加します。サンプル コードで、このサービスはアプリケーション間の制御権移転に使用されます。両方のサービスの特性を編集する必要がありません。
  - Attribute MTU size (bytes): **512**

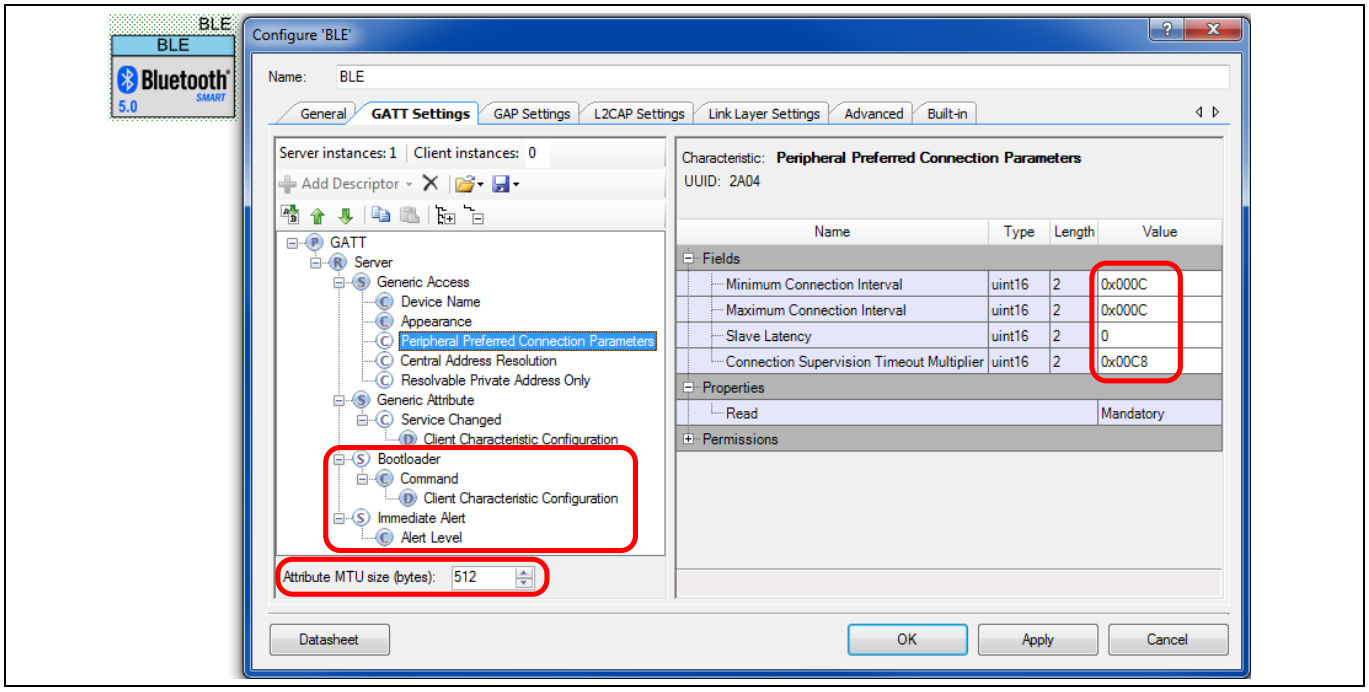


Figure 25　PSoC Creator の GATT Settings タブでの BLE コンポーネントの設定

- **GAP Settings** タブ
  - General, Device Name: サンプル コードは「BLE Bootloader」を使用します。アプリケーションに応じて異なる名称を使用できます。
  - Peripheral Configuration 0, Advertisement packet: **Local Name** をオンにし、**Complete** にセットします。
  - Security configuration 0 (**Figure 26** を参照してください), Security level: Unauthenticated pairing with encryption
  - Security configuration 0, I/O capabilities: **No Input No Output**
  - Security configuration 0, Bonding requirement: **No Bonding**

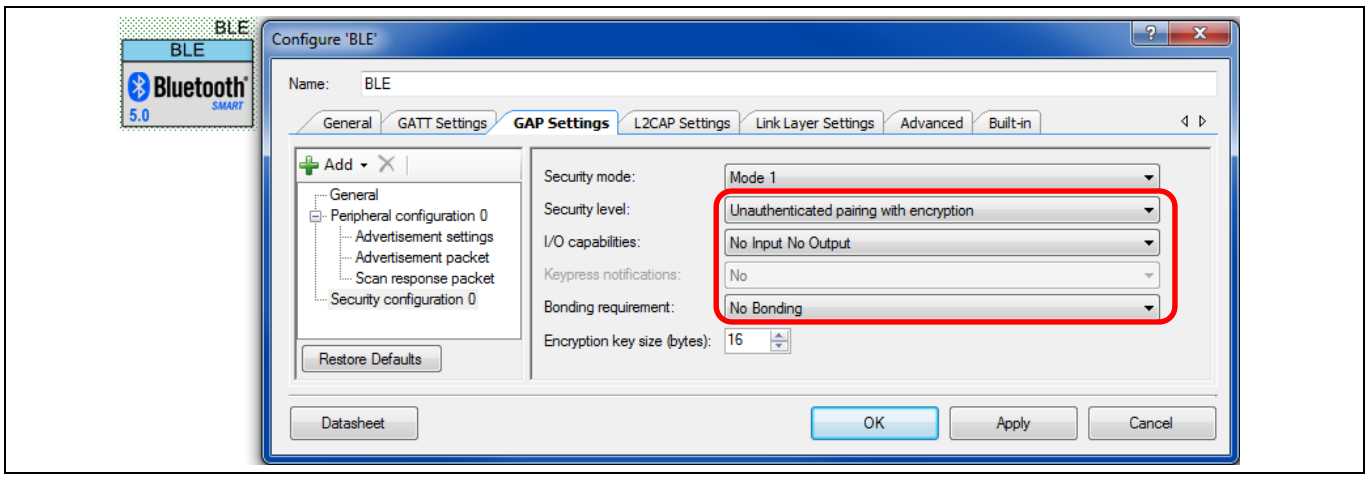


Figure 26　PSoC Creator の GAP Settings タブでの BLE コンポーネントの設定

- Link Layer Settings タブ

  - Link layer max TX and RX payload size (bytes): 251 (ブートローディングを速くするため)

3. **CE220959** の場合、App0 回路図に SMIF コンポーネントを追加し、SMIF コンポーネント コンフィギュレーション セクションに応じてコンポーネントを設定します。特に、デフォルト コンフィギュレーションから以下の変更を行ってください。

- (オプションであるが、CE220959 ファイルに対応するように推奨される) Name: SMIF
- **SMIF Datalines [2:3]**ボックスを選択します。外部メモリ IC は、PSoC 6 MCU デバイスとの Quad-SPI (4 データ ライン) インターフェースを備えています。**SMIF Datalines [0:1]**ボックスはデフォルトで選択されています。
- **Generate code from cy_smif.cysmif file** を非選択にします。*cy_smif_memconfig.h/.c* および *smif_mem.h/.c* ファイルは、サンプル コードと共に提供されています。

4. 必要に応じて、他のコンポーネント (LED およびボタン) を **CE216767** または **CE220959** Top Design 回路図からユーザーの Top Design 回路図にコピーします。
5. Design Wide Resources ウィンドウで

- **Pins** タブ: コンポーネント ピンを適切な物理ピンに接続します。BLE コンポーネントはピンを使用しないことに注意してください。
- **Clocks** タブ: WCO を有効にし、WCO から Clk_LF および Clk_Bak をソースします。これは、特定のモードで BLE コンポーネントを操作するために必要です。詳細については、BLE コンポーネントのデータシートを参照してください。
  - **CE220959** の場合、Clk_HF2 を有効にして、周波数を 50 MHz に設定します。これを行う最も簡単な方法は、Clk_HF2 セクションで 2 による除算オプションを選択することです。外部メモリインターフェースを正しく動作させるには、周波数制限が必要です。

6. プロジェクトの **Build Settings** ウィンドウで、App0 をブートローダに設定します。**Figure 10** を参照してください。**BLE** を通信チャネルに選択します。
7. **Build** > **Generate Application** を選択します。これにより、ブートローダ リンカー ファイル (例: *bootload_cm4.ld*) が作成されます。
8. プロジェクトの **Build Settings** ウィンドウに戻り、CM0+および CM4 の **Custom Linker Script** を対応するブートローダ リンカー スクリプト ファイルにセットします。**Figure 12** を参照してください。
9. *bootload_common.ld* で、フラッシュおよび RAM メモリ領域のサイズを以下のように編集します。

```
MEMORY
{
    flash_app0_core0  (rx)  : ORIGIN = 0x10000000, LENGTH = 0x10000
    flash_app0_core1  (rx)  : ORIGIN = 0x10010000, LENGTH = 0x30000
    flash_app1_core0  (rx)  : ORIGIN = 0x10040000, LENGTH = 0x32000
    flash_app1_core1  (rx)  : ORIGIN = 0x10072000, LENGTH = 0x02000
. . .
    ram_app0_core0    (rwx) : ORIGIN = 0x08000100, LENGTH = 0x1F00
    ram_app0_core1    (rwx) : ORIGIN = 0x08002000, LENGTH = 0x8000
. . .
    ram_app1_core0    (rwx) : ORIGIN = 0x08000100, LENGTH = 0x1FF00
    ram_app1_core1    (rwx) : ORIGIN = 0x08020000, LENGTH = 0x20000
. . .
```

デフォルトの *bootload_common.ld* は、各アプリケーションで各 CPU にフラッシュと RAM の同じメモリ量を割り当てます。サンプル コード用に割当てを調整しなければならない理由は以下のとおりです。

- BLE スタック サイズが大きい
- ブートローディング時間を短縮するために App1 サイズを最小限に抑える必要がある

サンプル コードで、App1 に対して同じ変更が行われることに注意してください。MDK および IAR リンカー スクリプト用に同様なメモリ レイアウトがあります。

10. *debug.h/.c*、*ias.h/.c* および *bootload_user.h/.c* ファイルを **CE216767** または **CE220959** フォルダから *app0.cydsn* フォルダにコピーアンドペーストします。既存のデフォルト *bootload_user.h/.c* ファイルを上書きします。また、CE220959 では、*cy_smif_memconfig.h/.c* および *smif_mem.h/.c* ファイルをコピーアンドペーストします。
- debug と ias (即時アラート サービス) ファイルはサンプル コードに特有のものであり、ユーザーのアプリケーションでは任意です。
- bootload_user ファイルは、デフォルトの UART トランスポート関数の代わりに BLE トランスポート関数を呼び出すように編集されました。
- CE220959 の場合、SMIF ファイルは外部メモリ コンフィギュレーションを定義し、外部メモリへのアクセスの関数を提供します。

11. PSoC Creator Workspace Explorer ウィンドウで、**Figure 27** に示すように、前のステップからのファイルをプロジェクト CM4 の *Header Files* と *Source Files* フォルダに移動します。
これらのファイルが *Shared Files* フォルダにある場合、CM0+ビルドに含まれ、コンパイル エラーが発生する可能性があります。
12. CM0+と CM4 のメイン コードを CE216767 と CE220959 からユーザーのメイン ファイルにコピーします。アプリケーションの必要に応じて、メイン ファイルおよびその他のソース ファイルを更新します。
13. app0 をビルドしてから、それをキットにプログラムします。
CE216767 または CE220959 からの app1 をインストールすることでユーザーの app0 をテストできます。サンプル コード ドキュメントの「Operation」セクションに記述の手順に従ってください。
14. ユーザー独自の app1 をビルドする際には、以下のファイルを編集してください。

- 3 つのリンカー スクリプト ファイル (**ステップ 9** および **4.3.2 節のステップ 6**)
- バッチ ファイル (**4.3.2 節のステップ 8**)

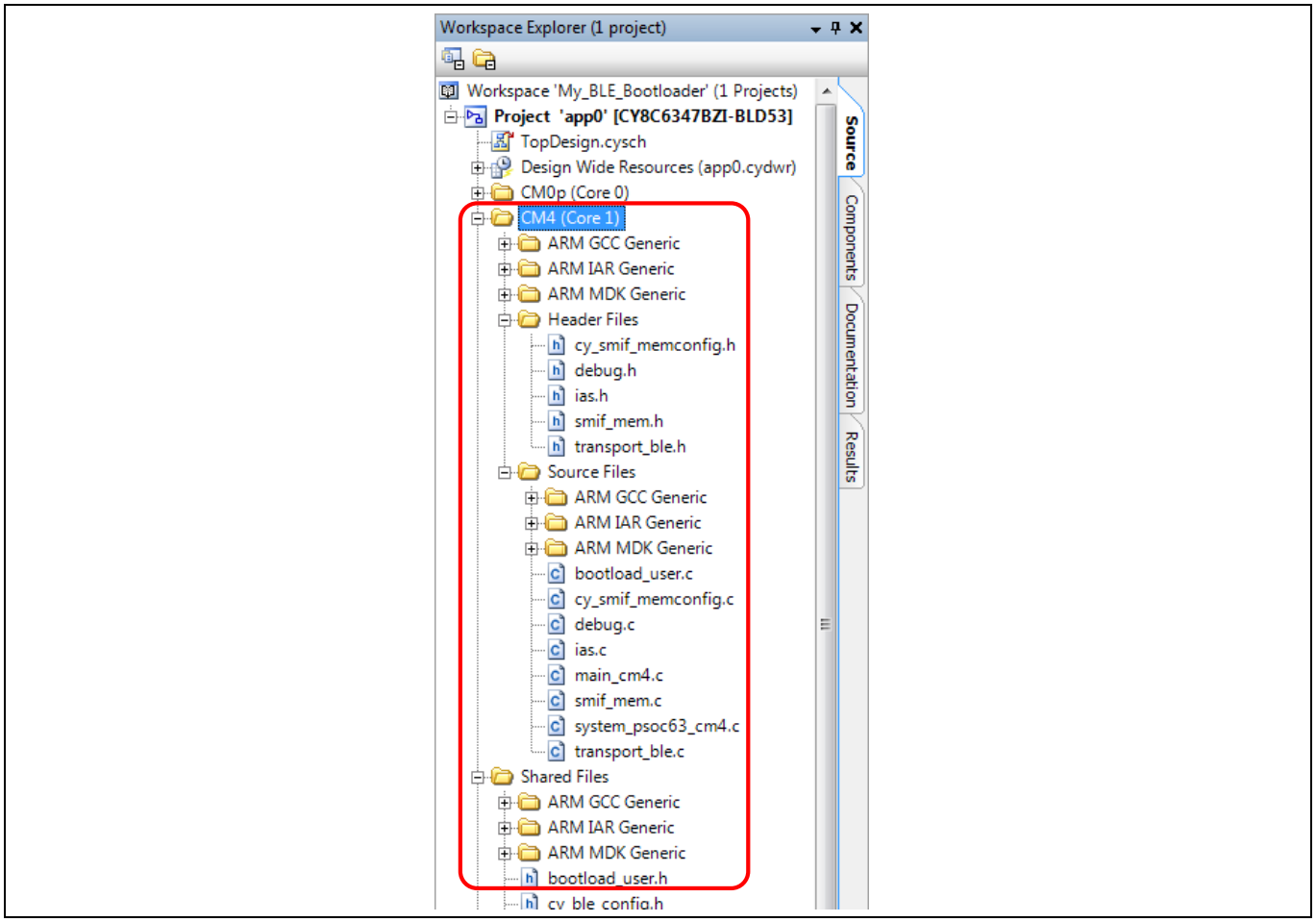


Figure 27 PSoC Creator CM4 フォルダに含まれるソース ファイル

5.3.1 アップグレードスタック付き BLE ブートローダ

このサンプル コードは、**Figure 28** に示すように 3 つのアプリケーションがあるという点で、他の **BLE ブートローダ サンプル コード**とは異なります。

この例では、App0 はブートローディングを行いません。単なるランチャー アプリケーションです。また、スタックの更新を一時的な場所から適切な場所にコピーし、スタックまたはユーザーアプリケーションを起動します。App0 は OTA ブートローディングでは更新できません。

App1 はスタックアプリケーションです。ブートローダと BLE スタックが含まれています。ブートローダは、ユーザーアプリケーションまたはスタックに更新をダウンロードできます。スタックを更新するには、

スタックがそれ自体を上書きできないため、一時的な場所で更新し、コピーのためにランチャーに切り替えます。

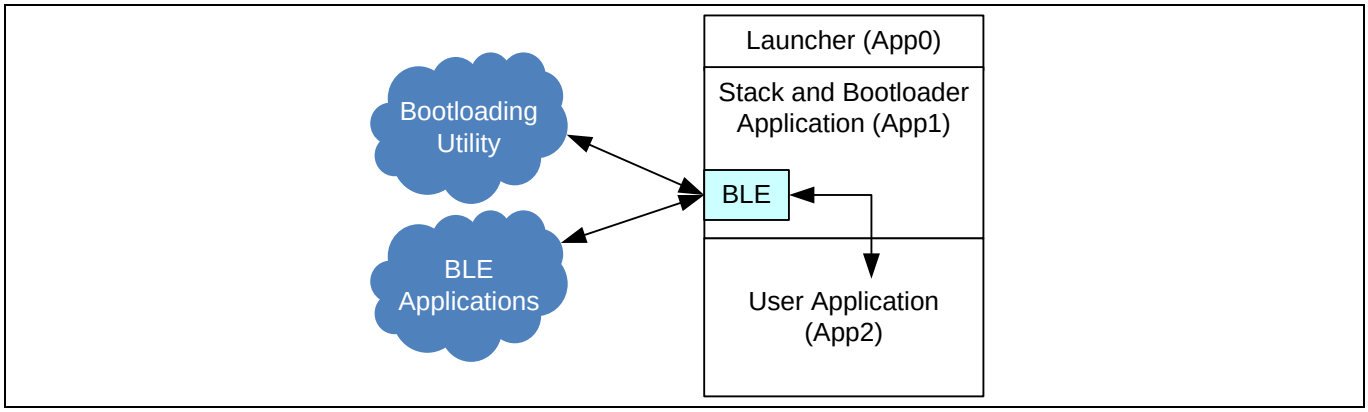


Figure 28　CE220960 のデータと制御フロー

App2 はユーザーアプリケーションです。PSoC Creator BLE コンポーネントを持ちますが、サポートするスタック コードのないプロファイルのみが含まれています。必要なスタック コードと変数はスタック アプリケーションと共有され、ユーザーアプリケーションのサイズを大幅に削減します。

このサンプル コードを作成する手順は、他の BLE サンプル コードの手順と似ています。主な違いは、3 つのプロジェクトを作成することと、BLE 構成が異なることです。

1. **Figure 23** に示すように、App0 とワークスペースを作成します。
2. 必要に応じて、コンポーネント (LED およびボタンなど) を **CE220960** App0 のトップデザイン回路図から使用するトップデザイン回路図にコピーします。
3. Design Wide Resources ウィンドウの **Pins** タブで、コンポーネント端子を適切な物理端子に接続します。手順については、**CE220960** とキットのガイドを使用してください。
4. プロジェクト **Build Settings** でブートローダとして App0 を構成します。(**Figure 10** を参照してください。アプリケーションのコピーと制御機能の転送を有効にするには、これを行う必要があります。) **Core** と **App type** ボックスのみを選択します。通信チャネルボックスは選択しないでください。
5. **Build** > **Generate Application** を選択します。これにより、*bootloader_cm4.ld* などのブートローダリンカファイルを含む多数のファイルが作成されます。
6. プロジェクトの **Build Settings** に戻り、CM0+および CM4 の **Custom Linker Script** をそれぞれのブートローダ リンカー スクリプト ファイルに設定します。**Figure 12** を参照してください。また、**CM4 User Commands** に *post_build_core1.bat* ファイルへの呼び出しを追加します。**Figure 13** を参照してください。
7. ファイルを更新し、**8** にリストされている新しいファイルを **CE220960** の対応するファイルからコピーします。アプリケーションの必要に応じてコードを追加します。

Table 5　　ブートローダ SDK のデフォルトから変更された CE220960 PSoC Creator App0 ファイル

ファイル	デフォルトから変更された内容
bootload_common.ld, bootload_cm0p.ld, bootload_cm4.ld *bootload_mdk_common.h, bootload_mdk_symbols.c* *bootload_cm0p.scat, bootload_cm4.scat*	3 つのアプリケーションでメモリ割り当てを変更
bootload_user.h, bootload_user.c	メタデータでのアプリケーション数を変更
main_cm0p.c, main_cm4.c	サンプルコード機能を実行するためのコードの追加

ファイル	デフォルトから変更された内容
新ファイル; サンプルコードからコピーし、共有ファイルフォルダに追加	
post_build_core1.bat	App1 および App2 のビルド場所へ出力をコピー

8. 新しいプロジェクト App1 をワークスペースに追加します。**他の BLE サンプルコード**の App0 と同じ方法で、プロジェクトにコンポーネントとファイルを追加して構成します。BLE 構成ダイアログを除き、**Over-The-Air bootloading with code sharing** の **Stack and Profile** を選択します。
9. ガイダンスとして CE220960 App1 を使用します。**CE220960** App1 には、コアごとに 1 つずつ、2 つのビルド後バッチファイルがあることに注意してください。
10. CE220960 の手順を使用してテストします。
11. 新しいプロジェクト App2 をワークスペースに追加します。コンポーネントとファイルを App2 に追加し、**他の BLE サンプルコード**の App0 と同じ方法で構成します。ただし、BLE 構成ダイアログを除き、**Over-The-Air bootloading with code sharing** の **Profile Only** を選択します。ガイダンスとして CE220960 App2 を使用してください。

5.4 PSoC 6 MCU のデュアルアプリケーション ブートローダ

本節では、サンプル コード **CE221984** の BLE ブートローダをビルドする方法を示します。本サンプル コードは、**基本ブートローダのサンプル コード**に似ていますが、1 つのダウンロード可能アプリケーションの代わりに 2 つのものがあります。アプリケーションは 3 つあります。App0 はブートローダ、App1 と App2 はダウンロード可能アプリケーションです。

ブートローダは、基本モードまたは工場出荷時のデフォルト (「ゴールデン イメージ」) モードでアプリケーションのいずれかに制御権を渡します。工場出荷時のデフォルト モードでは、ブートローダ (App0) はインストールされた有効な App1 を上書きしません。これをしようとすると、エラー メッセージが表示されます。基本モードでは、いずれかのアプリケーションも上書きできます。

本サンプル コードで、ブートローダは I^2C 通信チャネルを使用します。異なる通信チャネルに変更は簡単です。詳細は、**前の節**のいずれかを参照してください。

本サンプル コードをビルドするために、以下を行ってください。手順の詳細については、**PSoC 6 MCU の基本ブートローダ**または **SDK の使用方法**を参照してください。

1. **Figure 29** に示すように、App0 およびワークスペースを作成します。

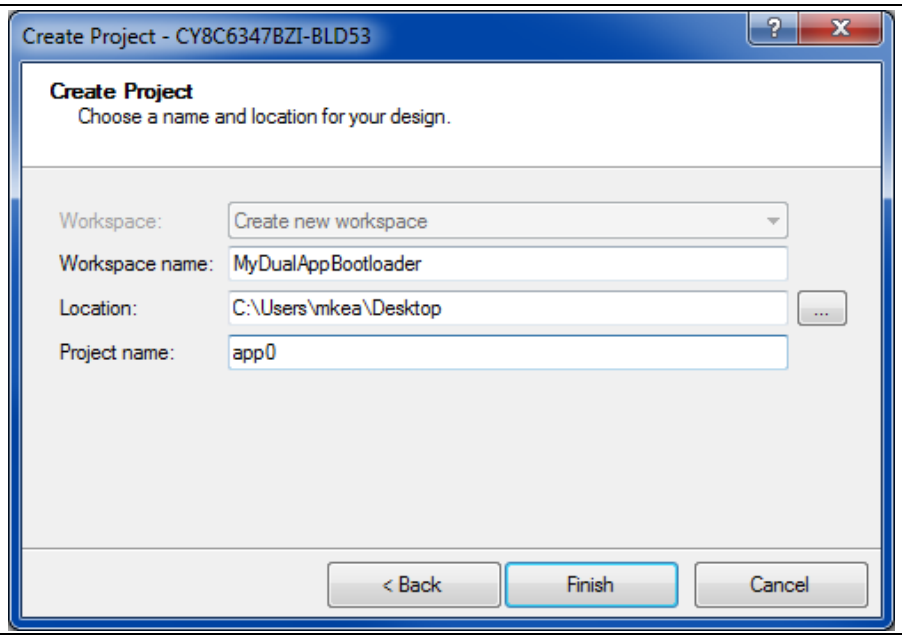


Figure 29　PSoC Creator のデュアルアプリケーション ブートローダ App0 を作成

2. プロジェクトの Build Settings ウィンドウで、App0 をブートローダに設定します。**Figure 10** を参照してください。**I2C** を通信チャネルに選択します。
3. **Build** > **Generate Application** を選択します。これにより、ブートローダ リンカー ファイル (例: *bootload_cm4.ld*) が作成されます。
4. プロジェクトの Build Settings ウィンドウに戻り、CM0+および CM4 の **Custom Linker Script** を対応するブートローダ リンカー スクリプト ファイルにセットします。**Figure 12** を参照してください。
5. App0 回路図に I2C コンポーネントを追加し、**Figure 18** に示すようにコンポーネントを設定します。ここで、**CE221984** で使用するピン コンポーネントを追加することもできます。このための最も簡単な方法は、サンプル コード回路図からコンポーネントをコピーアンドペーストすることです。
6. Design Wide Resources ウィンドウの **Pins** タブで、コンポーネント ピンを適切な物理ピンに接続します。
7. **8** にリストアップしたファイルを、**CE221984** での対応するファイルの内容で更新します。アプリケーションの必要に応じてコードを追加します。

Table 6　　ブートローダ SDK のデフォルトから修正された CE221984 App0 ファイル

ファイル	デフォルトからの修正内容
bootload_common.ld *bootload_mdk_common.h*	App2 用のメモリ領域の追加
bootload_user.h	工場出荷時のデフォルト イメージとして App1 を特定するために `CY_BOOTLOAD_GOLDEN_IMAGE_IDS` マクロをセット サンプル コードのモードを変更するために `CY_BOOTLOAD_OPT_GOLDEN_IMAGE` マクロを 0 以外の値に変更
bootload_user.c	UART から I^2C に変更されたトランスポート関数 工場出荷時のデフォルト (「ゴールデン イメージ」) チェックを行うために `Cy_Bootload_WriteData()` にコードを追加
main_cm0p.c *main_cm4.c*	サンプル コードの関数を実行するためのコードの追加

8. app0 プロジェクトをビルドし、それをターゲット キットにプログラムします。アプリケーション ダウンロード (ブートロード) プロセスをテストします。**CE221984** ドキュメントの「Operation」セクションに記載のテスト手順を使用できます。CY_BOOTLOAD_OPT_GOLDEN_IMAGE マクロを 0 以外の値に変更してみて、App1 をダウンロードする時のブートローダの異なる動作に注意してください。
ブートローダをテストするために、サンプル コードの既存の app1 と app2 プロジェクトを使用するか、または以下の手順を行ってユーザー独自のアプリケーションを作成できます。
9. app1 プロジェクトと app2 プロジェクトを追加します。両方のプロジェクトをインストール可能アプリケーションに設定できます。詳細な手順は **5.2.2 節のステップ 3** および**ステップ 4** に記述されています。主なタスクは以下の通りです。

- プロジェクトのビルド設定を更新します。
  - Peripheral Driver Library > Bootloader SDK:**Core** ボックスのみを選択します (**Figure 10**)。
  - カスタムのリンカー スクリプト (**Figure 12**)
  - ビルド後のバッチ ファイル (**Figure 13**)
- *Shared Files* フォルダ内のプロジェクトに *post_build_core1.bat* ファイルを追加します。標準的なバッチ ファイル内容は、**付録 E** を参照してください。
- プロジェクト回路図および Design Wide Resources のファイルを、**CE221984** での対応するファイルの内容で更新します。アプリケーションの必要に応じて設計を修正します。
- **Table 7** にリストアップしたファイルを、CE221984 での対応するファイルの内容で更新します。アプリケーションの必要に応じてコードを追加します。

Table 7 ブートローダ SDK のデフォルトから修正された CE221984 PSoC Creator App1 および App2 ファイル

ファイル	デフォルトからの修正内容
bootload_common.ld *bootload_mdk_common.h*	App2 用のメモリ領域の追加
bootload_cmp0.ld *bootload_cm4.ld* *bootload_cmp0.scat* *bootload_cm4.scat*	app1 または app2 用のメモリ領域およびアプリケーション ID を指定
bootload_user.h	工場出荷時のデフォルト イメージとして App1 を特定するために CY_BOOTLOAD_GOLDEN_IMAGE_IDS マクロをセットサンプル コードのモードを変更するために CY_BOOTLOAD_OPT_GOLDEN_IMAGE マクロを 0 以外の値に変更
main_cm0p.c *main_cm4.c*	サンプル コードの関数を実行するためのコードの追加

5.5 PSoC 6 MCU 暗号化ブートローダ

このセクションでは、サンプル コード **CE222802** に示すように、セキュリティ機能を備えたブートローディングシステムを構築する方法を示します。このサンプル コードは、**basic bootloader code sample** に似ていますが、安全なブートロードに必要な追加機能 (署名と暗号化) があります。App0 はブートローダです。ダウンロード可能な App1 は署名, 暗号化, またはその両方が可能です。

このサンプル コードでは、ブートローダは UART 通信チャネルを使用します。別の通信チャネルへの変更は簡単です。詳細については、**前のセクション**のいずれかを参照してください。

このサンプル コードをビルドするには、次の手順を実行します。これらの手順の詳細については、**PSoC 6 MCU Basic Bootloaders** または **SDK の使用方法**を参照してください。

Note: *署名と暗号化にはキーが必要です。サンプル コードではデフォルトのキーを含むファイルが提供されていますが、アプリケーションには新しいキーを作成する必要があります。サンプル コードは新しいキーの作成をサポートしていますが、そのためにはまずコンピュータに OpenSSL と Python をインストールする必要があります。詳細については、サンプル コードのドキュメントを参照してください。*

1. **Figure 30** に示すように、App0 とワークスペースを作成します。

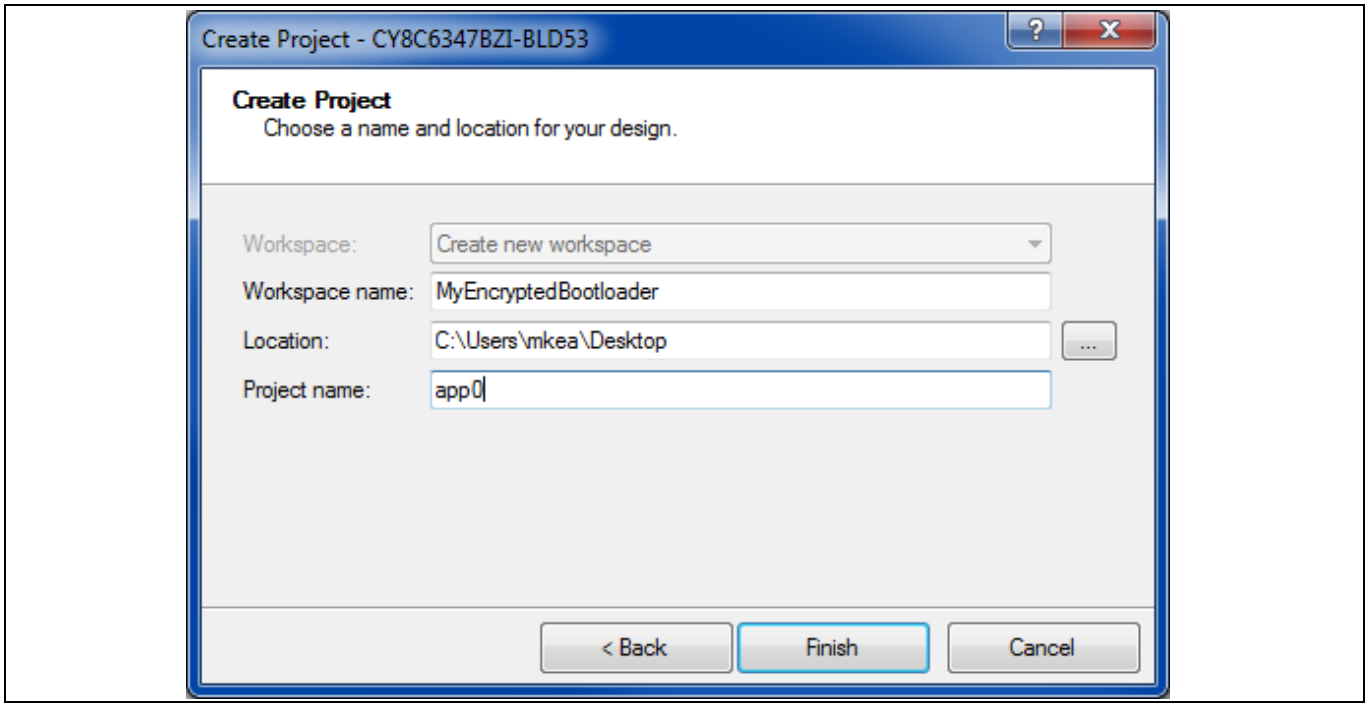


Figure 30 PSoC Creator の暗号化ブートローダ App0 を作成

2. プロジェクトのビルド設定で App0 をブートローダとして構成します。**Figure 10** を参照してください。通信チャネルに **UART** を選択します。
3. また、ビルド設定で **Peripheral Driver Library** > **PDL** > **crypto** を選択し、**Variant** で **Extra library** を選択して AES 復号化のサポートを有効にします。
4. **Build** > **Generate Application** を選択します。 これにより、*bootloader_cm4.ld* などのブートローダ リンカーファイルが作成されます。
5. プロジェクトの Build Setting に戻り、CM0+および CM4 の **Custom Linker Script** をそれぞれのブートローダリンカースクリプトファイルに設定します。**Figure 12** を参照してください。また、**CM4 User Commands** に *post_build_core1.bat* ファイルへの呼び出しを追加します。**Figure 13** を参照してください。
6. UART コンポーネントを App0 回路図に追加し、**Figure 18** に従ってコンポーネントを構成します。この時点で、**CE222802** が使用する Pin Components も追加できます。これを行う最も簡単な方法は、サンプル コードの回路図から Components をコピーして貼り付けることです。
7. Design-Wide Resources ウィンドウの **Pins** タブで、Component 端子を適切な物理端子に接続します。手順については CE222802 とキットのガイドを使用してください。
8. ファイルを更新し、**Table 11** にリストされている新しいファイルを **CE222802** の対応するファイルからコピーします。アプリケーションの必要に応じてコードを追加します。

Table 8　ブートローダ SDK のデフォルトから変更された CE222802 App0 ファイル

ファイル	デフォルトからの修正内容
bootload_common.ld *bootload_mdk_common.h*	異なった署名サイズ
bootload_user.h	暗号ブロックと安全な検証を有効にするため #defines を設定
bootload_user.c	キー変数と復号化機能の追加
main_cm0p.c main_cm4.c	サンプル コード機能を実行するためのコードの追加
新ファイル; サンプルコードからコピーし、共有ファイルフォルダに追加	
cy_si_config.h *cy_si_keystorage.h, .c*	キーストレージ
post_build_core1.bat	署名と出力ファイルの暗号化

9. **CE222802** からプロジェクトフォルダ *CE222802_Keys.cylib* をワークスペース フォルダにコピーします。このフォルダには、*post_build_core1.bat* によって使用されるキーファイルが含まれています。これらのキーを独自のキーに置き換えられます。手順については、サンプル コードのドキュメントを参照してください。
10. このフォルダには、PSoC Creator ライブラリプロジェクトファイル *CE222802_Keys.cylib* も含まれています。オプションで、このライブラリプロジェクトをワークスペースに追加できます。
11. app0 プロジェクトをビルドし、ターゲットキットにプログラムします。アプリケーションのダウンロード (ブートロード) プロセスをテストします。サンプル コードドキュメントの Operation セクションにリストされているテスト手順を使用できます。

Note: *ブートローダプロジェクトは、デバイスの初期化の一環として、PSoC 6 MCU 監視フラッシュ (Sflash) のフラッシュブートモジュールによって検証されるような形式で構築されます。app0 を別のプロジェクトに置き換えるようにデバイスを再プログラムすると、検証は失敗し、デバイスは起動しません。他のプロジェクトでデバイスを再プログラムする場合は、最初に app0 の検証を無効にしてからデバイスに再プログラムする必要があります。サンプル コードのドキュメントの手順を参照してください。*

サンプル コードにすでにある app1 プロジェクトを使用してブートローダをテストするか、次の手順に示すように独自のアプリケーションを作成できます。

12. app1 プロジェクトを追加します。設定可能なのはインストール可能なアプリケーションです。必要な詳細な手順は、**5.2.2 節のステップ 3** および**ステップ 4** に示されています。重要なタスクは次のとおりです。

- プロジェクトビルド設定の更新
  - Peripheral Driver Library > Bootloader SDK。Core ボックスのみを選択 (**Figure 10**)。
  - リンカー カスタム スクリプト (**Figure 12**)
  - ビルド後のバッチ ファイル (**Figure 13**)
- プロジェクトの回路図とデザイン全体のリソースファイルを、**CE222802** の対応するファイルの内容で更新します。アプリケーションの必要に応じて設計を変更します。
- ファイルを更新し、CE222802 の対応するファイルから **Table 12** にリストされている新しいファイルをコピーします。アプリケーションの必要に応じてコードを追加します。

Table 9　　ブートローダ SDK のデフォルトから変更された CE221984 App1 ファイル

ファイル	デフォルトからの修正内容
bootload_common.ld *bootload_mdk_common.h*	メモリ領域を指定
bootload_cmp0.ld *bootload_cm4.ld* *bootload_cmp0.scat* *bootload_cm4.scat*	メモリ領域とアプリ ID を指定
bootload_user.h	暗号ブロックと安全な検証を有効にするため `#defines` を設定
main_cm0p.c *main_cm4.c*	サンプル コード機能を実行するためのコードの追加
新ファイル; サンプルコードからコピーし、共有ファイルフォルダに追加	
cy_si_config.h	キーストレージ
post_build_core1.bat	署名と出力ファイルの暗号化

6 関連資料

PSoC 6 MCU リソースの包括的なリストについては、サイプレスコミュニティの **KBA223067** を参照してください。

アプリケーション ノート	
AN221774 – Getting Started with PSoC 6 MCU	PSoC 6 MCU デバイスと、最初の ModusToolbox または PSoC Creator プロジェクトの構築方法について説明
AN210781 – Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity	BLE 接続を備えた PSoC 6 MCU デバイスおよび初めての PSoC Creator プロジェクトのビルド方法について説明
AN215656 – PSoC 6 MCU: Dual-CPU system Design	PSoC 6 MCU のデュアル CPU アーキテクチャおよびシンプルなデュアル CPU 設計のビルド方法について説明
AN219434 – Importing PSoC Creator Code into an IDE for a PSoC 6 MCU Project	PSoC Creator によって生成されたコードを所望の IDE へエクスポートする方法を説明

サンプル コード	
CE213903 – PSoC 6 MCU Basic DFU	PSoC 6 MCU 用の UART、I^2C および SPI DFU を説明
CE216767 – PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity Bootloader	PSoC 6 MCU 用の BLE ブートローダを説明
CE220959 – PSoC 6 MCU BLE Bootloader with External Memory	BLE ブートローダと同様であるが、ダウンロードされたアプリケーションは外部メモリに一時的に保存されてから、最終保存場所にコピーされる
CE220960 – PSoC 6 MCU BLE Bootloader with Upgradeable Stack	BLE ブートローダと同様であるが、BLE スタックはアプリケーションに加えて更新可能
CE221984 – PSoC 6 MCU Dual-Application I2C Bootloader	基本的な I^2C ブートローダと同様であるが、1 つのダウンロード可能アプリケーションの代わりに 2 つのものを管理
CE222802 – PSoC 6 MCU Encrypted Bootloader	基本的な UART ブートローダと同様であるが、アプリケーションはデジタル署名および暗号化される

PSoC Creator コンポーネント データシート	
UART	UART ベースの通信に対応
I2C	I^2C ベースの通信に対応
SPI	SPI ベースの通信に対応
BLE	BLE 通信に対応

デバイス資料	
PSoC 6 MCU: PSoC 63 with BLE Datasheet	**PSoC 6 MCU: PSoC 63 with BLE Architecture Technical Reference Manual**

開発キット資料	
CY8CKIT-062-BLE	PSoC 6 BLE Pioneer Kit
CY8CKIT-062-WiFi-BT	PSoC 6 WiFi-BT Pioneer Kit
CY8CPROTO-063-BLE	PSoC 6 BLE Prototyping Kit
CY8CPROTO-062-4343W	PSoC 6 Wi-Fi Prototyping Kit

ツール資料	
ModusToolbox	ModusToolbox IDE は、IoT デザイナーの開発を簡素化します。Windows, macOS, および Linux 用の使いやすいツールと使い慣れたマイクロコントローラ (MCU) 統合開発環境 (IDE) を提供します。

関連資料

PSoC Creator	PSoC Creator は、ハードウェアとファームウェアの同時編集、PSoC デバイスのコンパイルとデバッグを可能にします。アプリケーションは、スケマティックキャプチャと、150 以上の検証済みの製品対応の周辺コンポーネントを使用して作成されます。download タブで Quick Start と User Guides を確認してください。
Peripheral Driver Library (PDL)	PSoC Creator 4.2 によってインストールされます。User Guides と API リファレンスについては、<PDL インストールフォルダ>/doc をご覧ください。

7 付録 A. DFU ホスト ツール

DFU ホストツール (DHT) は、ModusToolbox IDE に同梱されたスタンドアローンのグラフィカル ツールです。(PSoC Creator に同梱されるブートローダホストプログラム (BHP) もあります。これは DHT とほぼ同様に動作します。) このツールは、DFU がインストールされた PSoC デバイスとの通信に使用されます。このツールの使用で、以下のことが可能です。

- デバイスへのアプリケーションのダウンロードおよびインストール
- デバイスにすでにインストールされたアプリケーションの検証
- デバイスからのアプリケーションの消去

Note: *DHT を使用して DFU アプリケーションをデバイスにインストールすることはできません。その代わりに、ModusToolbox IDE や Cypress Programmer などの他のツールを使用して、デバイス SWD / JTAG ポートを介してプログラムする必要があります。DFU アプリケーションのインストール後、DHT を使用してダウンロード可能なアプリケーションをインストールできます。*

Figure 31 に示すように、DHT は UART、I^2C、SPI または USB 経由のサイプレス MCU との通信に対応しています。接続に利用可能なすべてのデバイスが表示されます。UART および USB の場合、適切なケーブルを接続することでコンピュータから通信を直接行えます。I^2C および SPI の場合、KitProg モジュールなどの特殊な通信ポートが必要です。選択したポートによって、ポート コンフィギュレーション フィールドが異なります。

Note: *現時点で、PSoC 6 MCU は UART、I^2C および SPI ベースの DFU にのみ対応しています。*

Note: *DHT は Bluetooth Low-Energy (BLE) に対応していません。その代わりに、サイプレスの **CySmart** 製品を使用してください。CySmart ユーザー ガイドの「Updating Peripheral Device Firmware」セクションを参照してください。*

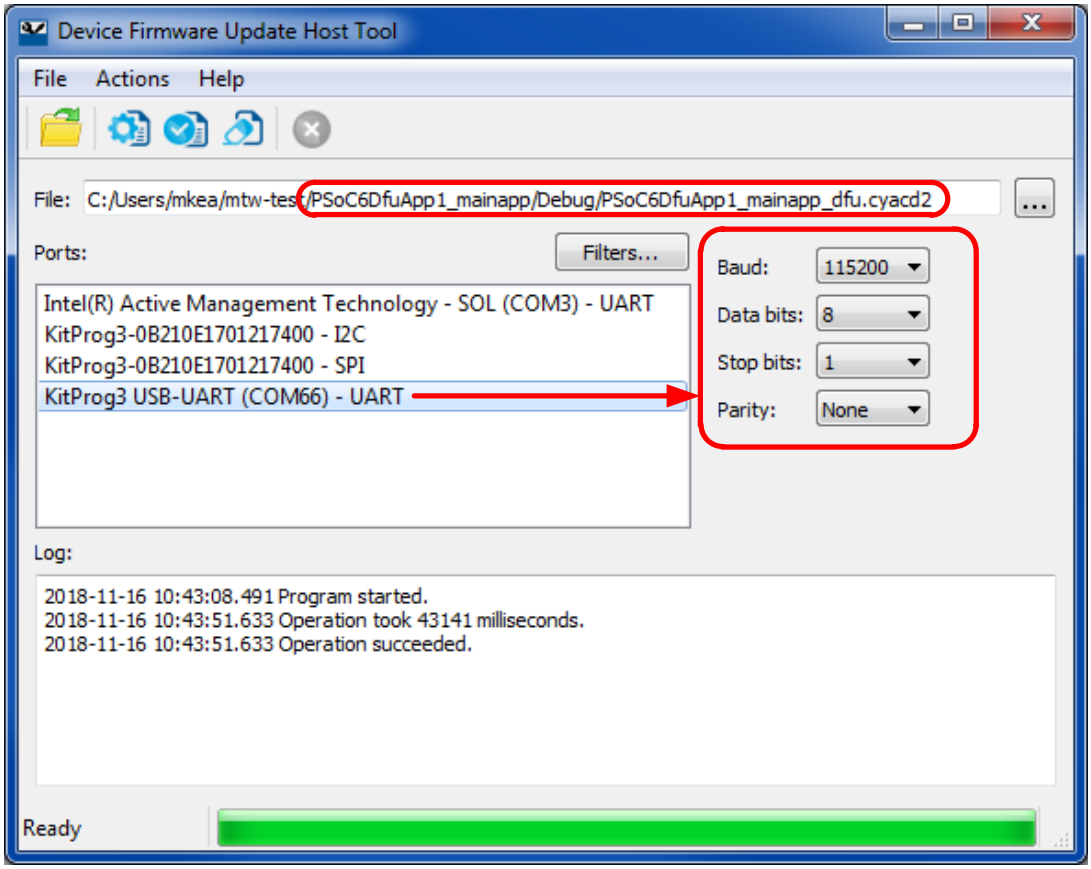


Figure 31 DFU ホスト プログラム グラフィカル ユーザー インターフェース (GUI)

8 付録 B. ホスト コマンド／応答プロトコル

DFU モジュールは、使用される通信チャネルにかかわらず、簡単なコマンド応答プロトコルを使用してホストと通信します。DFU モジュールは通信チャネルからコマンドを受信し、通信チャネルに 1 バイト以上を送信することでそれぞれのコマンドに応答します。**Figure 2** を参照してください。

コマンドおよび応答は、送信されるデータの完全性を保証するようにパケット化されたバイト ストリームの形式です。パケット妥当性チェック方式が含まれており、2 の補数の 16 ビット チェックサムから成ります。

8.1 コマンド／応答パケットの構造

ホストから DFU モジュールに送信される通信パケットの構造は **Figure 32** のようです。

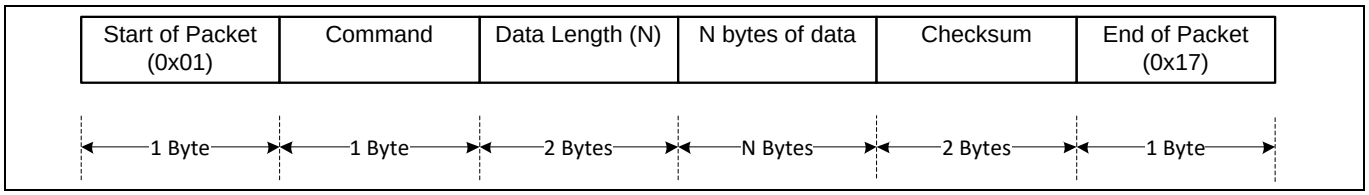


Figure 32 DFU コマンド パケットの構造

ブートローダからホストに送信される応答パケットの構造は **Figure 33** のようです。

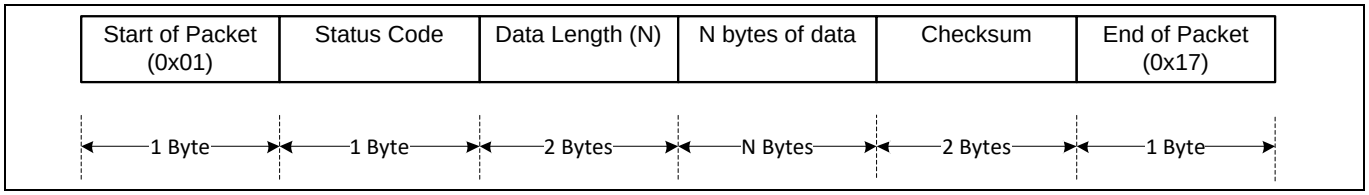


Figure 33 DFU 応答パケットの構造

すべてのマルチバイトのフィールドは、LSB ファーストで送信されます。

8.2 コマンド

Table 10 に、DFU SDK によってサポートされるすべてのコマンドの一覧を示します。DFU 終了を除くすべてのコマンドは、DFU 開始コマンドが受信されるまで無視されます。

Table 10 DFU コマンド一覧

DFU コマンド		
開始／終了	**DFU 動作**	**その他**
DFU 開始	**データ送信**	**アプリケーション検証**
DFU 同期化	**応答無しのデータ送信**	**アプリケーション メタデータ設定**
DFU 終了	**データ プログラミング**	**メタデータ取得**
	データ検証	**EIVector 設定**
	データ消去	

コマンド実行時間には特定の要件がありません。

Table 11 に、DFU SDK によってサポートされるステータスおよびエラー コードの一覧を示します。

Table 11 DFU ステータスおよびエラー コード一覧

ステータス／エラーコード	値	説明
CY_DFU_SUCCESS	0x00	コマンドは正常に受信され、実行された
CY_DFU_ERROR_VERIFY	0x02	書き込みが失敗した後の不揮発性メモリ (NVM) の検証
CY_DFU_ ERROR_LENGTH	0x03	送信されるデータ量が期待より大きい
CY_DFU_ ERROR_DATA	0x04	データが適切な形式でない
CY_DFU_ ERROR_CMD	0x05	コマンドが認識できない
CY_DFU_ ERROR_CHECKSUM	0x08	パケット チェックサムまたは CRC が期待された値と一致しない
CY_DFU_ ERROR_ROW	0x0A	フラッシュ行の番号が有効でない
CY_DFU_ ERROR_ROW_ACCESS	0x0B	フラッシュ行番号へアクセスできない (例: MPU 保護のため)
CY_DFU_ERROR_UNKNOWN	0x0F	未知のエラーが発生した

8.2.1 DFU 開始

ブートロード動作を開始します。DFU 終了を除くすべてのコマンドは、このコマンドが受信されるまで無視されます。デバイス情報と DFU SDK バージョンで応答します。

入力

- コマンド バイト: 0x38
- データ バイト
  - 4 バイト (任意): 製品 ID。これらのバイトが含まれ、00 00 00 00 でない場合、デバイス製品 ID データと比較されます。

出力

- ステータス／エラー コード
  - 成功
  - エラー コマンド
  - エラー データ (製品 ID 不一致に使用)
  - エラー長さ
  - エラー チェックサム
- データ バイト
  - 4 バイト: デバイス JTAG ID
  - 1 バイト: デバイス レビジョン
  - 3 バイト: DFU SDK バージョン

8.2.2 DFU 同期化

新しいコマンドを受け付けられる既知の状態に DFU をリセットします。バッファ内のデータは、すべて廃棄されます。このコマンドは、DFU モジュールとホストが互いに同期しなくなる場合にのみ必要です。

入力

- コマンド バイト: 0x35
- データ バイト: 該当無し

出力: 該当無し – このコマンドが認識されない

8.2.3 DFU 終了

DFU から抜け出します。DFU 動作を終了します。

入力

- コマンド バイト: 0x3B
- データ バイト: 該当無し

出力: 該当無し – このコマンドが認識されない

8.2.4 データ送信

DFU モジュールにデータ ブロックを転送します。このデータは、データ プログラミングまたはデータ検証コマンドを見越してバッファリングされます。一連のデータ送信コマンドが送信される場合、データは前のブロックに付随されます。このコマンドは、いくつかのプロトコルでのチャネル欠乏を防ぐために大きいデータの転送を複数の小さいデータに分割するために使用されます。

入力

- コマンド バイト: 0x37
- データ バイト
  - n バイト: 書き込みまたは検証用データ

出力

- ステータス／エラー コード
  - 成功
  - エラー コマンド
  - エラー データ
  - エラー長さ
  - エラー チェックサム
- データ バイト: 該当無し

8.2.5 応答無しのデータ送信

DFU モジュールが応答を生成しないことを除き、データ送信コマンドと同じです。いくつかのアプリケーションで DFU 時間を減らします。

入力

- コマンド バイト: 0x47
- データ バイト:
  - n バイト: 書き込みまたは検証用データ

出力: 該当無し

8.2.6 データ プログラミング

デバイスの内部フラッシュの 1 行または外部不揮発性メモリ (NVM) の 1 ページにデータを書き込みます。一連のデータ送信または応答無しのデータ送信コマンドの後に続く場合があります。

入力

- コマンド バイト: 0x49
- データ バイト
  - 4 バイト: アドレス。正しいメモリ アドレス空間内にあり、適切に整列される必要があります。内部フラッシュでは、フラッシュ行境界に整列しなければなりません。外部メモリでは、外部メモリのデータ整列要件を満たさなければなりません。
  - 4 バイト: 書き込みデータ全体の CRC-32C。データはプログラミングの前と後の両方で検証されます。
  - n バイト - フラッシュの行または外部 NVM ページに書き込むデータ

出力

- ステータス／エラー コード
  - 成功
  - エラー コマンド
  - エラー データ
  - エラー長さ
  - エラー チェックサム
  - エラー フラッシュ行
  - エラー フラッシュ行アクセス
- データ バイト: 該当無し

8.2.7 データ検証

デバイスの内部フラッシュの 1 行または SMIF の 1 ページとデータを比較します。一連のデータ送信または応答無しのデータ送信コマンドの後に続く場合があります。

このコマンドは任意であり、その有無は *dfu_user.h* 内のユーザー コンフィギュレーション マクロに依存します。

入力

- コマンド バイト: 0x4A
- データ バイト
  - 4 バイト: アドレス。正しいメモリ アドレス空間内にあり、適切に整列される必要があります。内部フラッシュでは、フラッシュ行境界に整列しなければなりません。外部メモリでは、外部メモリのデータ整列要件を満たさなければなりません。
  - 4 バイト: 検証データ全体の CRC-32C。
  - n バイト: フラッシュの行または SMIF ページと比較されるデータ

出力

- ステータス／エラー コード
  - 成功
  - エラー検証

  - エラー コマンド
  - エラー データ
  - エラー長さ
  - エラー チェックサム
  - エラー フラッシュ行
  - エラー フラッシュ行アクセス
- データ バイト: 該当無し

実装の詳細

- このコマンドは、すべてのデータ バイトが指定されたフラッシュ アドレスから始まるバイトと一致した場合は「成功」ステータス コード、そうでない場合は「エラー検証」を返します。

8.2.8 データ消去

指定されたフラッシュ メモリ列の内容を消去

このコマンドは任意であり、その有無は *dfu_user.h* 内のユーザー コンフィギュレーション マクロに依存します。

入力

- コマンド バイト: 0x44
- データ バイト
  - 4 バイト: アドレス。正しいメモリ アドレス空間内にあり、適切に整列される必要があります。内部フラッシュでは、フラッシュ行境界に整列しなければなりません。外部メモリでは、外部メモリのデータ整列要件を満たさなければなりません。

出力

- ステータス／エラー コード
  - 成功
  - エラー コマンド
  - エラー データ
  - エラー長さ
  - エラー チェックサム
  - エラー フラッシュ行
  - エラーフラッシュ行アクセス
- データ バイト: 該当無し

8.2.9 アプリケーション検証

フラッシュまたは外部 NVM 内のアプリケーション用チェックサムが有効であるかを報告します。

入力

- コマンド バイト: 0x31
- データ バイト
  - 1 バイト: 検証されるアプリケーションの番号。0～[アプリケーション数 − 1] の範囲内です。

出力

- ステータス／エラー コード
  - 成功
  - エラー コマンド
  - エラー データ
  - エラー長さ
  - エラー チェックサム
  - エラー フラッシュ行アクセス
- データ バイト
  - 1 バイト: アプリケーションが有効な場合は 1、有効でない場合は 0

8.2.10 アプリケーション メタデータ設定

特定のアプリケーションのメタデータを設定します。「**10 付録 D. アプリケーション メタデータ**」を参照してください。

Note: *メタデータが変わらないようにユーザーが DFU SDK を設定した場合、このコマンドはメタデータを更新しません。*

入力

- コマンド バイト: 0x4C
- データ バイト
  - 1 バイト: アプリケーション番号
  - 8 バイト: **10** によるメタデータ フィールド フォーマット

出力

- ステータス／エラー コード
  - 成功
  - エラー コマンド
  - エラー長さ
  - エラー データ
  - エラー チェックサム
  - エラーフラッシュ行アクセス
- データ バイト: 該当無し

8.2.11 メタデータ取得

選択されたメタデータ バイトを報告します。

このコマンドは任意であり、その有無は *dfu_user.h* 内のユーザー コンフィギュレーション マクロに依存します。

入力

- コマンド バイト: 0x3C
- データ バイト
  - 2 バイト: 行内のオフセット最小値。0〜511

  - 2 バイト: 行内のオフセット最大値。0〜511 (0 と 511 を含む)

出力

- ステータス／エラー コード
  - 成功
  - エラー コマンド
  - エラー長さ
  - エラー データ
  - エラー チェックサム
  - エラーフラッシュ行アクセス
- データ バイト
  - N バイト – オフセットの最小値と最大値のバイトごと (両方を含む)

8.2.12 EIVector 設定

暗号化初期ベクタ (EIV) を設定します。これは、フラッシュにデータを書き込む前に DFU モジュールがデータを復号することを可能にます。

このコマンドは任意であり、その有無は *dfu_user.h* 内のユーザー コンフィギュレーション マクロに依存します。

入力

- コマンド バイト: 0x4D
- データ バイト
  - n バイト: ベクタ。0 または 8 または 16 バイト。リトルエンディアン式の生データ

出力

- ステータス／エラー コード
  - 成功
  - エラー コマンド
  - エラー長さ
  - エラー データ
  - エラー チェックサム
- データ バイト: 該当無し

9 付録 C. .cyacd2 ファイル フォーマット

.cyacd2 ファイルはダウンロード可能アプリケーション データを含みます。CyMCUElfTool によって作成され、**Figure 5** に示すように、ターゲット DFU モジュールにアプリケーションを送信するためにサイプレスの**付録 A. DFU ホスト ツール**や **CySmart** などのホスト プログラムによって使用されます。ファイル データは Intel 16 進形式と同様な ASCII 16 進数です。各データ バイトは 2 つの文字で表されます。例えば、0x1E バイトは 0x31 (ASCII での「1」) および 0x45 (ASCII での「E」) で表されます。

すべてのマルチバイト フィールドはリトルエンディアン式です。

ファイルは一連のライン (行) から成ります。各行は ASCII CR、LF 文字で終わります。行は以下のタイプのいずれかです。

- 暗号化初期ベクタ: 暗号化初期ベクタ行は`@EIV:<bytes>`のフォーマットです。<bytes> のデータは、DFU モジュールへの **SetEIV コマンド**内でホスト プログラムによって使用されます。
- アプリケーション検証情報: アプリケーション検証情報は以下のフォーマットです。

  ```
  @APPINFO:[__cy_app_verify_start],[__cy_app_verify_length]
  ```

  開始と長さのデータは、DFU モジュールへの**アプリケーション メタデータ設定コマンド**内でホスト プログラムによって使用されます。
- ヘッダ: ヘッダ行の構造は **Figure 34** のようです。

ファイル バージョン	シリコン チップ ID	シリコン チップ レビジョン	チェックサム タイプ	アプリケーション ID	製品 ID
1 バイト	4 バイト	1 バイト	1 バイト	1 バイト	4 バイト

Figure 34 .cyacd2 ヘッダ行の構造

  - ファイル バージョン: 1 から番号付けられます。
  - シリコン チップ ID、シリコン チップ レビジョン、製品 ID: アプリケーションが誤ったデバイスからダウンロードされないようにします。
  - チェックサム タイプ: DFU パケットを検証する方式です (**8 付録 B. ホスト コマンド／応答プロトコル**を参照してください)。0 = チェックサム、1 = CRC。
  - アプリケーション ID: **Figure 6** を参照してください。これはまた、どのアプリケーション メタデータの部分が更新されるかを制御します。
- データ: データ行の構造は **Figure 35** のようです。

N の値は、一連の**データ送信**または**応答無しのデータ送信**コマンドの後に**データ プログラミング**または**データ検証**のコマンドを実行することで送信する総データ量に等しいです。

N の値は通常、NVM 行の長さに等しいですが、必要ではありません。例えば RAM へのダウンロード時、N は任意の値です。

ヘッダ	アドレス	データ
1 文字: 「:」	4 バイト	N バイト

Figure 35 .cyacd2 データ行の構造

10 付録 D. アプリケーション メタデータ

DFU SDK は、アプリケーションに関する情報を格納するために NVM (場合によっては RAM) の指定された領域を使用します。**Figure 6** を参照してください。一般的に、メタデータ情報は以下の目的で使用されます。

- アプリケーションの検証
- アプリケーションからアプリケーションへの制御権移転
- 一時的な場所から指定された場所へのアプリケーション イメージのコピー

Figure 6 に示したように、メタデータは通常、フラッシュの 1 行または NVM の 1 ページを占めます (フラッシュ容量の小さいデバイスでは、複数の行やページを使用する場合があります)。**Figure 6** はまた、メタデータはアプリケーションの外に配置されることを示します。

Table 12 は、DFU メタデータの場所、サイズおよび使用法を定義するシンボルを示します。シンボルは、SDK リンカー スクリプト ファイルおよび C ソース ファイルで定義されています。

Note: *示すすべての例は GCC コンパイラとリンカー用のものです。同様なステートメントが MDK および IAR コンパイラとリンカー用のソース ファイルとリンカー スクリプト ファイルにあります。*

Table 12 メタデータ関連のシンボル

シンボル	定義するファイル	用途
`flash_boot_meta`	*dfu_cm0p.ld, dfu_cm4.ld*	メタデータを格納する物理的なメモリ領域を定義
使用例: `flash_boot_meta (rw) : ORIGIN = 0x100FFA00, LENGTH = 0x400` PSoC 6 MCU の 1MB の (0x1000 0000 で始まる) ユーザー フラッシュの最後の 1KB で領域を定義		
`__cy_boot_metadata_addr` `__cy_boot_metadata_length`	*dfu_cm0p.ld, dfu_cm4.ld*	メタデータを格納するための、コンパイラから独立したメモリ アドレス範囲を定義。DFU SDK コードで使用されており、ユーザー コードでされる場合がある

使用例:
`/* Bootloader SDK metadata limits */`
`/* Note that __cy_memory_0_row_size equals the row length in bytes of PSoC 6 user flash. */`
`__cy_boot_metadata_addr = ORIGIN(flash_boot_meta);`
`__cy_boot_metadata_length = __cy_memory_0_row_size;`		

シンボル	定義するファイル	用途
.cy_boot_metadata	*bootload_cm0p.ld* *bootload_cm4.ld*	このセクションに DFU メタデータが格納される。ビルド時、CyMCUElfTool はこのセクションのチェックサムを計算し、セクションの最後の 4 バイトに配置。チェックサムが不要な場合、セクションの名前を変更してください

使用例:

```
.cy_boot_metadata :
{
    KEEP(*(.cy_boot_metadata))
} > flash_boot_meta
```

CY_DFU_MAX_APPS	*dfu_user.h*	ユーザーが DFU メタデータでサポートされる最大のアプリケーション数を制御することを可能にする

使用例:

```
/* The smallest metadata size is CY_DFU_MAX_APPS * 8 bytes per app +
   an optional 4 bytes for metadata checksum */
#define  CY_DFU_MAX_APPS  (2u)
```

CY_DFU_METADATA_WRITABLE	*dfu_user.h*	DFU は必ずしもメタデータを書き込む必要はない。それは、アプリケーションやユーザー コードによって行われることがある。あるいは、メタデータはアプリケーション内のコンパイル時定数によりセットされる場合がある。NVM 行より小さいメタデータを持つアプリケーションがある。すべての場合、DFU がメタデータを書き込まないように、このマクロを 0 にセット。ただしブートローダには、メタデータ領域が存在し、適切に初期化されたことが必要。これは DFU それ自身またはアプリケーションによって行われる

使用例:

```
/* A non-zero value allows writing metadata  with the SetAppMetadata command. */
#define  CY_DFU_METADATA_WRITABLE  (1)
```

10.1 メタデータの構造

Figure 36 に示すように、アプリケーション メタデータはアプリケーションごとの 8 バイトおよびそれに続くチェックサムの 4 バイトから成ります。

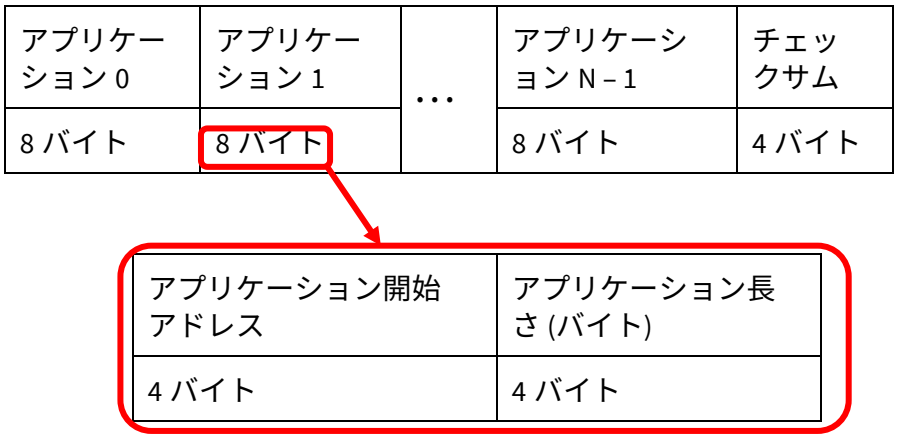


Figure 36 メタデータの構造

アプリケーションの数 N は *dfu_user.h* ファイルでセットすることがあります:

```
#define DFU_MAX_APPLICATIONS (N)
```

N のデフォルト値は 2 です。

各アプリケーション開始アドレスはフラッシュ行または NVM ページの境界に整列する必要があります。アプリケーション長さはフラッシュ行または NVM ページの長さの倍数でなければなりません。

チェックサムは、**データ プログラミング**および**データ検証**の DFU コマンドで使用される同じアルゴリズムで計算されます。デフォルトのアルゴリズムは CRC-32C です。

11 付録 E. ビルド後のファイル一覧

以下には、**CE213903** からの App1 用のビルド後の bash およびバッチ ファイルの一覧を示します。ファイルは、他のサンプル コードと同じでない場合は類似しています。同じ bash/バッチは複数のダウンロード可能なアプリケーションに使用できます (例えば、**デュアルアプリケーション ブートローダ**の app2)。

11.1 ModusToolbox のビルド後の Bash ファイル

```
#!/bin/bash

########################################################################################################################
# This script is designed to post process a PSoC 6 application. It performs sign and merge.
#
# usage:
#   dfu_postbuild.bash <MCUELFTOOL_LOC> <CM0P_LOC> <CM4_LOC> <MCU_CORE>
#
########################################################################################################################

MCUELFTOOL_LOC=$1
CM0P_LOC=$2
CM4_LOC=$3
MCU_CORE=$4

echo Script: cymcuelftool_postbuild
echo 1: MCUELFTOOL_LOC : $MCUELFTOOL_LOC
echo 2: CM0P_LOC       : $CM0P_LOC
echo 3: CM4_LOC        : $CM4_LOC
echo 4: MCU_CORE       : $MCU_CORE
echo

filenameNoExt_cm0p="${CM0P_LOC%.*}"
filenameNoExt_cm4="${CM4_LOC%.*}"

if [ "$MCU_CORE" == "ARM_CM4" ]; then
$MCUELFTOOL_LOC --sign $CM4_LOC --output $filenameNoExt_cm4"_signed.elf"
$MCUELFTOOL_LOC --merge $filenameNoExt_cm4"_signed.elf" $filenameNoExt_cm0p"_signed.elf" --output $filenameNoExt_cm4"_final.elf"
```

```
#DFU
$MCUELFTOOL_LOC --sign $filenameNoExt_cm4"_final.elf" CRC --output $filenameNoExt_cm4"_dfu.elf"
$MCUELFTOOL_LOC -P $filenameNoExt_cm4"_dfu.elf" --output $filenameNoExt_cm4"_dfu.cyacd2"

else
$MCUELFTOOL_LOC --sign $CM0P_LOC --output $filenameNoExt_cm0p"_signed.elf"
  if [ -e $filenameNoExt_cm4"_signed.elf" ]; then
  $MCUELFTOOL_LOC --merge $filenameNoExt_cm4"_signed.elf" $filenameNoExt_cm0p"_signed.elf" --output $filenameNoExt_cm4"_final.elf"
  fi
fi
```

11.2 PSoC Creator のビルド後のバッチ ファイル

```
@rem Usage:
@rem Call post_build_core1.bat <tool> <output_dir> <project_short_name>
@rem E.g. in PSoC Creator 4.2:
@rem      post_build_core1.bat creator ${OutputDir} ${ProjectShortName}

@echo ----------------------------------------
@echo   Post-build commands for Cortex-M4 core
@echo ----------------------------------------

@rem Set proper path to your PDL 3.x and above installation
@set PDL_PATH="C:\Program Files (x86)\Cypress\PDL\3.0.1"

@set CY_MCU_ELF_TOOL=%PDL_PATH%"\tools\win\elf\cymcuelftool.exe"

@set IDE=%1

@if "%IDE%" == "creator" (
    @set OUTPUT_DIR=%2
    @set PRJ_NAME=%3
    @set ELF_EXT=.elf
)

@if "%IDE%" == "uvision" (
```

```
    @set OUTPUT_DIR=%2
    @set PRJ_NAME=%3
    @set ELF_EXT=.axf
)

@if "%IDE%" == "iar" (
    @set OUTPUT_DIR=%2
    @set PRJ_NAME=%3
    @set ELF_EXT=.out
)

@if "%IDE%" == "eclipse" (
    @set OUTPUT_DIR=%2
    @set PRJ_NAME=%3
    @set ELF_EXT=
)

%CY_MCU_ELF_TOOL% -S %OUTPUT_DIR%\%PRJ_NAME%%ELF_EXT% CRC
%CY_MCU_ELF_TOOL% -P %OUTPUT_DIR%\%PRJ_NAME%%ELF_EXT% --output %OUTPUT_DIR%\%PRJ_NAME%.cyacd
```

改訂履歴

Document version	Date of release	Description of changes
**	2018-07-05	これは英語版 002-13924 Rev. *D を翻訳した日本語版 Rev. **です。
*A	2019-12-23	これは英語版 002-13924 Rev. *F を翻訳した日本語版 Rev. *A です。
*B	2021-05-12	テンプレートの変更を実施。 これは英語版 002-13924 Rev. *G を翻訳した日本語版 Rev. *B です。

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2021-05-12
Published by
Infineon Technologies AG
81726 Munich, Germany

© 2021 Infineon Technologies AG. All Rights Reserved.

Do you have a question about this document?
Go to www.cypress.com/support

Document reference
002-23478 Rev. *B

重要事項

本文書に記載された情報は、いかなる場合も、条件または特性の保証とみなされるものではありません（「品質の保証」）。本文に記された一切の事例、手引き、もしくは一般的価値、および／または本製品の用途に関する一切の情報に関し、インフィニオンテクノロジーズ（以下、「インフィニオン」）はここに、第三者の知的所有権の不侵害の保証を含むがこれに限らず、あらゆる種類の一切の保証および責任を否定いたします。

さらに、本文書に記載された一切の情報は、お客様の用途におけるお客様の製品およびインフィニオン製品の一切の使用に関し、本文書に記載された義務ならびに一切の関連する法的要件、規範、および基準をお客様が遵守することを条件としています。

本文書に含まれるデータは、技術的訓練を受けた従業員のみを対象としています。本製品の対象用途への適合性、およびこれら用途に関連して本文書に記載された製品情報の完全性についての評価は、お客様の技術部門の責任にて実施してください。

本製品、技術、納品条件、および価格についての詳しい情報は、インフィニオンの最寄りの営業所までお問い合わせください（**www.infineon.com**）。

警告事項

技術的要件に伴い、製品には危険物質が含まれる可能性があります。当該種別の詳細については、インフィニオンの最寄りの営業所までお問い合わせください。

インフィニオンの正式代表者が署名した書面を通じ、インフィニオンによる明示の承認が存在する場合を除き、インフィニオンの製品は、当該製品の障害またはその使用に関する一切の結果が、合理的に人的傷害を招く恐れのある一切の用途に使用することはできないこと予めご了承ください。