参考資料

*Application Note*

# *C2000 デバイスでのセキュア ブート*

*Pramod Prabhakara, Karthik Rajakumar, Christopher Chiarella, Matt Kukucka, and Ronak Harsora*

概要

本アプリケーション レポートでは、F2838x デバイスに搭載されたセキュア フラッシュ ブート機能を使用し、実際のコード実行前にブート コード認証という追加のセキュリティ層を適用して、フラッシュからアプリケーションをブートする方法を説明します。本ドキュメントは、F28003x、F280013x、F280015x、F28P55x、F28P551x、F28P65x のデバイス ファミリにも適用されます。

## 目次

**1** 概要....2
**2** セキュア フラッシュ ブートの概要....2
**3 CMAC** 認証....3
**4** セキュア フラッシュ ブート オプション....3
**5** セキュア フラッシュ ブートのフロー....4
**6 C2000Ware** のサンプルの詳細....5
**7 16KB** を超えるフラッシュ コードの認証....7
**8** デバッグ リソース....8
**9** 追加情報および考慮事項....8
**10 C2000 CMAC** アルゴリズムの **OpenSSL** への整合....10
10.1 C28x メモリおよびバイナリ ファイルのバイト順序....10
10.2 フラッシュ バイナリのバイト順序....10
10.3 CMAC キーのバイト順序....11
10.4 CMAC 出力の整列手順....11
10.5 動作例....12
10.6 相違点のまとめ....13
**11** 参考資料....13
**12** 改訂履歴....14

## 図の一覧

図 3-1. CMAC 動作....3
図 6-1. CMAC 用に Hex ユーティリティを有効にした CPU1 のプロパティ例....6

## 表の一覧

表 2-1. デバイス全体におけるセキュア フラッシュ ブートの概要....2
表 4-1. セキュア フラッシュ ブート モードの構成詳細....3
表 8-1. セキュア フラッシュ ブートのデバッグ シナリオ....8
表 10-1. CMAC キーのバイト順序の対応....11
表 10-2. C2000 と OpenSSL の CMAC フローの比較....13

## 商標

C2000™ is a trademark of Texas Instruments.
Arm® is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
すべての商標は、それぞれの所有者に帰属します。


Copyright © 2026 Texas Instruments Incorporated

# 1 概要

TMS320F2838x は、産業用モーター ドライブ、ソーラー インバータおよびデジタル電源、電気自動車および輸送機器、センシングおよび信号処理などの高度な閉ループ制御アプリケーション用に設計された強力な 32 ビット浮動小数点リアルタイム マイコンです。本デバイスは、デュアルコアの C28x アーキテクチャに加えて、重要な通信タスクをオフロードする新しいコネクティビティ マネージャ (CM) をサポートしており、システム性能を大幅に向上させます。これらのリアルタイムマイコンのハードウェア効率は、強力なアプリケーションを実現しますが、具体的で独自性のあるアプリケーションを生み出すのはコードです。本デバイスには、安全なソリューションの構築を支援するためのデュアル コード セキュリティ モジュール (DCSM) が搭載されており、これに含まれるセキュア ブート機能によって、不正な更新によるアプリケーションの実行を防ぐ機能が強化されます。

DCSM は、デュアル セキュリティ ゾーンのコンセプトを採用しており、フラッシュのワン タイム プログラマブル (OTP) 領域にプログラムされたセキュリティ構成によって、保護可能なリソース (フラッシュ セクタや RAM) のゾーン 1 / ゾーン 2 / 非セキュアへの割り当てが決まります。これに加えて、ゾーン 1 / ゾーン 2 リソースを EXEONLY として構成することもできます。これにより、その領域からのコード実行のみが許可されます。

詳細については、『TMS320F2838x テクニカル リファレンス マニュアル』の「DCSM」章を参照してください [1]。

# 2 セキュア フラッシュ ブートの概要

アプリケーション フラッシュ ブートに関連する DCSM 機能の 1 つは、実行前にフラッシュ内のユーザー アプリケーション コードを認証できることです。これにより、フラッシュ メモリにアプリケーション コードがプログラムされた後に確実に改ざんされないようにして、その整合性を確保します。ゾーン 1 EXEONLY フラッシュ セクタに適用する場合、この機能は重要なユーザー アプリケーション コードに対する追加のセキュリティ層として機能します。セキュア フラッシュ ブート機能は、従来のフラッシュ ブート オプションに加えて、追加のブート オプション群を提供します。

セキュア フラッシュ ブートは、128 ビットの AES-CMAC 認証アルゴリズムを使用して実現されています。このアルゴリズムは、アプリケーション コードの内容に対して実行され、合否ステータスを返し、認証が成功した場合にのみアプリケーション コードの実行に進みます。表 2-1 に、デバイスのさまざまなサブシステムにおけるこの機能の概要を示します。各 CPU サブシステムの BootROM は、そのサブシステムにおけるアプリケーション コードの先頭 16KB の認証を開始します。この処理をライマリ セキュア ブートと呼びます。各 CPU サブシステムにおける先頭 16KB 以降のアプリケーション コードの認証を、拡張セキュア ブートと呼びます。これは、事前認証済みのアプリケーション コードから任意で開始することもできます。

セキュア フラッシュ ブート中に CMAC 認証アルゴリズムが実行されるため、ブート アップ シーケンスでは、通常 (非セキュア) のフラッシュ ブートと比較し、ユーザー アプリケーションに到達するまでに追加の時間が必要になります。デバイスの CM コアのセキュア フラッシュ ブートは、CPU1 や CPU2 のセキュア フラッシュ ブート実装と比べて、所要時間が短くなることに注意してください。これは、CM がハードウェア AES アクセラレータを使用しているためです。

**表 2-1. デバイス全体におけるセキュア フラッシュ ブートの概要**

| サブシステム (コア) | セキュア ブート機能 | CMAC アルゴリズムの実装 | フラッシュ ブート コード先頭 16KB の認証に要する追加時間 |
|---|---|---|---|
| CPU1 SS (C28_1) | あり | ソフトウェア (セキュア ROM ユーティリティ) + AES ROM テーブル | 約 400ms (INTOSC 10MHz 動作時) |
| CPU2 SS (C28_2) | あり | ソフトウェア (セキュア ROM ユーティリティ) + AES ROM テーブル | 約 20ms (PLL 200MHz 動作時) |
| CMSS (CM4) | あり | ソフトウェア (セキュア ROM ユーティリティ) + ハードウェア AES アクセラレータ | 約 6ms (PLL 125MHz 動作時) |


English Document: SPRACT3
Copyright © 2026 Texas Instruments Incorporated

# 3 CMAC 認証

暗号ベース メッセージ認証コード (CMAC) は、入力データ ブロックから認証タグを構築する、AES ベースの認証アルゴリズムです。入力データ ブロックは、128 ビットの CMAC キーとともに、128 ビットずつ暗号エンジン / ソフトウェア (CPU サブシステム ベース) に供給されます。キーは、CPU1 USER OTP の 0x78018 ～ 0x7801F に格納されており、デバイスのすべてのコアで CMAC 認証アルゴリズムに使用されます。暗号エンジン / ソフトウェアは、入力データ ブロック全体に対して CMAC アルゴリズムを実行し、最終的な 128 ビット認証タグを生成します。

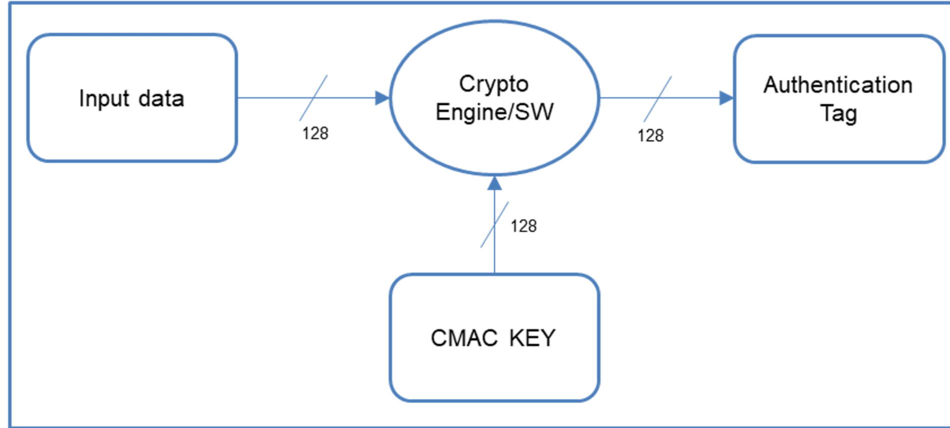


**図 3-1. CMAC 動作**

# 4 セキュア フラッシュ ブート オプション

表 4-1 に、3 つのコアで使用可能なさまざまなプライマリ セキュア フラッシュ ブート オプションと、それに対応する構成を示します。アプリケーション コードによって開始される拡張セキュア ブートについては、セクション 7 で説明しています。詳細については、『TMS320F2838x テクニカル リファレンス マニュアル』の「ROM コードおよびペリフェラル ブート」章を参照してください [1]。

**表 4-1. セキュア フラッシュ ブート モードの構成詳細**

| セキュア ブート オプション (1) | BOOTDEFx/ BOOTMODE 値 (2) | フラッシュ エントリ ポイント (3) | CPU1/CPU2 エントリ アドレス | CPU1/CPU2 における 128 ビット ゴールデン CMAC タグの位置 | CM エントリ アドレス | CM における 128 ビット ゴールデン CMAC タグの位置 |
|---|---|---|---|---|---|---|
| オプション 0 | 0x0A | セクタ 0 | 0x00080000 | 0x00080002 | 0x00200000 | 0x00200004 |
| オプション 1 | 0x2A | セクタ 4 | 0x00088000 | 0x00088002 | 0x00210000 | 0x00210004 |
| オプション 2 | 0x4A | セクタ 8 | 0x000A8000 | 0x000A8002 | 0x00250000 | 0x00250004 |
| オプション 3 | 0x6A | セクタ 13 | 0x000BE000 | 0x000BE002 | 0x0027C000 | 0x0027C004 |

(1) セキュア ブート オプションは、CPU1/CPU2/CM 用に個別に選択できます。
(2) CPU1 の場合、BOOTDEFx フィールドは、CPU1 USER OTP メモリ領域の Zx-BOOTDEF-LOW / Zx-BOOTDEF-HIGH の一部です。CPU2/CM の場合、BOOTMODE フィールドは、CPU1 のアプリケーション コードによってそれぞれ設定される CPU1TOCPU2IPCBOOTMODE/CPU1TOCMIPCBOOTMODE レジスタの一部です。
(3) セキュア ブート機能はゾーン 1 にのみ適用可能であるため、選択したフラッシュ セクタをゾーン 1 EXEONLY として構成する必要があります。また、セクタ サイズに関係なく、プライマリ セキュア フラッシュ ブートは、選択したセクタの最初の 16KB に対してのみ動作します。たとえば、セクタ 4 とセクタ 8 はそれぞれ 64KB ですが、最初の 16KB のみが考慮されます。


Copyright © 2026 Texas Instruments Incorporated

## 5 セキュア フラッシュ ブートのフロー

デバイスにおけるセキュア フラッシュ ブートの実装は、次の 2 ステップのプロセスで構成されます。

1. **認証タグの生成** – イメージ作成時にデバイスの外部で行われます。
   a. C2000™ または Arm®、hex ユーティリティは、入力 CMACKEY と、ゴールデン CMAC 認証タグのメモリ領域を保持する CMAC アプリケーション データ構造を使用して、フラッシュ ブート コード イメージに対して CMAC アルゴリズムを実行します。hex ユーティリティの詳細については、[3] および [4] を参照してください。
   b. 生成されたゴールデン CMAC タグは、hex ファイルの指定された位置 (表 4-1 参照) に組込まれます。
   c. hex イメージ (ゴールデン CMAC タグを含む) は、対応するフラッシュのセクタにプログラムされます。
   d. 適切なセキュア フラッシュ ブート モードは、表 4-1 に従って選択され、CPU1 USER OTP にプログラムされます。

2. **フラッシュ内のアプリケーション ブート コードの認証** – セキュア フラッシュ ブート実行の一部として、デバイス内で行われます
   a. BOOTDEFx/BOOTPINCONFIG フィールドは、表 4-1 に従ってセキュア フラッシュ ブート オプションを選択するように構成されており、リセット直後にデバイスはブートし、指定されたフラッシュ セクタに対して CMAC アルゴリズムを実行します。
   b. CMAC アルゴリズムによって生成されたタグは、事前にプログラムされた位置にあるゴールデン CMAC タグと比較されます。
   c. タグの照合に成功すると、ブート プロセスは認証されたフラッシュ コードに分岐し、実行を開始します。
   d. タグの照合に失敗すると、CPU1 / CPU2 / CM で次のような異なるアクションが実行されます。
      i. CPU1 の場合、デバイスはリセットされます (コードはループ内に維持され、ウォッチドッグのタイムアウト時に XRSn が自動的に発行されます)。
      ii. CPU2 の場合、セキュア ブート失敗フラグが CPU2TOCPU1IPCBOOTSTS レジスタに設定され、セキュア フラッシュ CMAC エラー コードを含む IPC コマンドが CPU1 に送信されます。その後、CPU2 ブート コードは、CPU1 が必要なアクションを実行するまでループで待機します。CPU2TOCPU1IPCBOOTSTS レジスタのコピーは、CPU2 のアドレス 0x0000 0002 の位置にもキャプチャされます。
      iii. CM の場合、セキュア ブート失敗フラグが CMTOCPU1IPCBOOTSTS レジスタに設定され、セキュア フラッシュ CMAC エラー コードを含む IPC コマンドが CPU1 に送信されます。その後、CM ブート コードは、CPU1 が必要なアクションを実行するまでループで待機します。CMTOCPU1IPCBOOTSTS レジスタのコピーは、CM のアドレス 0x2000 0000 の位置にもキャプチャされます。

---

**注**

CMAC アルゴリズムは、イメージの認証タグ計算時、およびイメージの認証時においても、ゴールデン タグが格納されているメモリ アドレスをすべて 1 として扱います。

---



English Document: SPRACT3
Copyright © 2026 Texas Instruments Incorporated

## 6 C2000Ware のサンプルの詳細

C2000Ware [2] には、セキュア フラッシュ ブート用のアプリケーション設定を示すサンプルが用意されています。このサンプルには、各コア用のセキュア フラッシュ ブート アプリケーション プロジェクトが含まれています。このサンプルでは、セキュア フラッシュ ブート エントリ アドレス +16KB を超えるフラッシュ コードを認証する方法についても詳細に説明しています。このカスタム フラッシュ範囲認証機能の詳細については、セクション 7 で説明しています。このサンプルでは、認証されるフラッシュ セクタがゾーン 1 EXEONLY として事前設定されいることを前提としており、認証にはデフォルトの CMACKEY を使用します。CPU1 USER OTP でのカスタム CMACKEY およびその他の DCSM 設定のプログラミングの詳細については、[1] および [2] を参照してください。

**C2000Ware** の場所:<C2000Ware_Install_Directory>/driverlib/f2838x/examples/c28x/boot

プロジェクト名:

- boot_ex1_cpu1_cpu2_cm_secure_flash_cpu1
- boot_ex1_cpu1_cpu2_cm_secure_flash_cpu2
- boot_ex1_cpu1_cpu2_cm_secure_flash_cm

含まれるファイル:

- ソース ファイル – メイン アプリケーション コードを含む
  - 例:boot_ex1_cpu1_cpu2_cm_secure_flash_cpu1.c
- HEX リンカ コマンド ファイル – c2000 または arm hex ユーティリティに、フラッシュ メモリ全体の長さに関する詳細を提供
  - 例:boot_ex1_flash_hex_lnk_cpu1.cmd
- CMAC キー テキスト ファイル – c2000 または arm hex ユーティリティに、ユーザーの CMAC キーを提供
  - 例:boot_ex1_user_cmac_key.txt
  - cmac_key フォーマットの詳細については、[3] および [4] を参照してください。

サンプルの実行方法:

1. アプリケーションを CPU1 フラッシュにロードします (CPU2 および CM アプリケーションについても同様)。
   a. *.out ファイルではなく、*.hex ファイルをロードします。
2. 接続解除し、CPU1 にのみ再接続します。
3. ブート アップ時にセキュア フラッシュ ブートを実行するようデバイスを設定するには:
   a. エミュレーション ブート (サンプル / 開発時に推奨)
      i. CCS メモリ ウィンドウで、BOOTPINCONFIG の位置 (0x0D00) を 0x5AFFFFFF に、BOOTDEF の位置 (0x0D04) を 0x0000000A に設定します。
   b. スタンドアロン ブート (デプロイ時に推奨)
      i. BOOTPINCONFIG および BOOTDEF に対応する CPU1 USER OTP 領域をプログラムします。詳細については、[1] を参照してください。
4. CCS を通じて CPU1 をリセットし、「resume」(再開) をクリックします。
5. controlCARD の LED を観察し、成功しているかを確認します。
   a. 3 つのコアすべてが正常にセキュア ブートし、フラッシュ メモリの内容全体を認証すると、3 つの LED (各コアに 1 つ) が点滅します。
6. controlCARD を使用しない場合、CPU サブシステムに基づいて、以下の GPIO トグルを確認します。
   a. CPU1 – GPIO31
   b. CPU2 – GPIO34
   c. CM - GPIO145


Copyright © 2026 Texas Instruments Incorporated

ゴールデン **CMAC** タグ生成のためのアプリケーション コード要件:

セキュア フラッシュ ブート オプション **0** における、**CPU1 / CPU2** のゴールデン **CMAC** タグのメモリ アロケーション

```
// Implementation for CPU1/CPU2
#pragma RETAIN(cmac_sb_1)
#pragma LOCATION (cmac_sb_1, 0x080002)
const char cmac_sb_1[8] = {0};
```

セキュア フラッシュ ブート オプション **0** における、**CM** のゴールデン **CMAC** タグのメモリ アロケーション

```
#pragma RETAIN(cmac_sb_1)
#pragma LOCATION (cmac_sb_1, 0x00200004)
const uint8_t cmac_sb_1[16] = {0};
```

定数の char 型または unsigned 整数型の定義により、ゴールデン CMAC タグ用のメモリが割り当てられます。詳細については、セクション 6 の例を参照してください。

- 変数名は、cmac_sb_1、cmac_sb_2、cmac_sb_3、cmac_sb_4 のいずれかである必要があります。
  - C28x の詳細については、『TMS320C28x アセンブリ言語ツール ユーザー ガイド』で確認できます [3]。
  - CM の詳細については、『ARM アセンブリ言語ツール ユーザー ガイド』で確認できます [4]。
- LOCATION プラグマを使用して、認証対象範囲内における CMAC ゴールデン タグ用のアドレスを指定します。CPU1 / CPU2 の場合、このアドレスはエントリ アドレス + 2 である必要があり、CM の場合、このアドレスはエントリ アドレス + 4 である必要があります。
- 変数を 0 に初期化したままにします。

HEX ユーティリティを使用するためのアプリケーション設定:

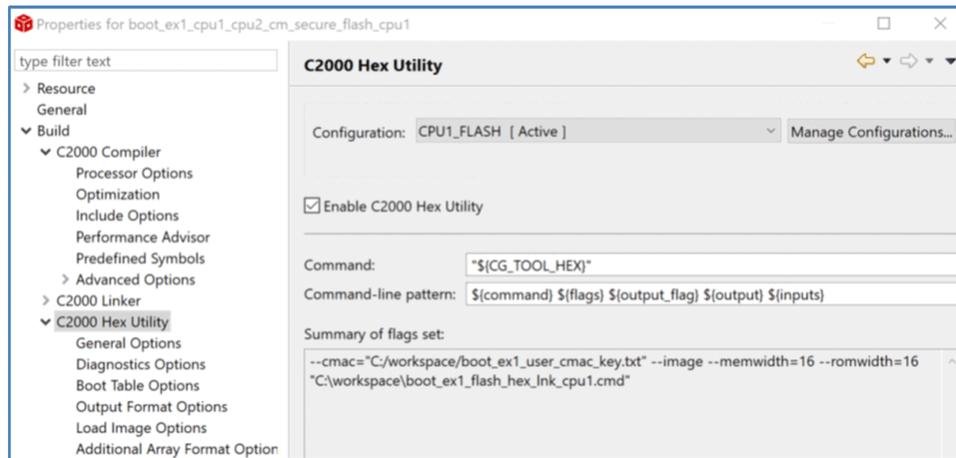


**図 6-1. CMAC 用に Hex ユーティリティを有効にした CPU1 のプロパティ例**

各コア プロジェクトで、ゴールデン CMAC タグを生成するために hex ユーティリティを有効化します (図 6-1 参照)。フラグには以下が含まれます。

- 「--cmac」は、ユーザー CMAC キー テキスト ファイルへのパスを提供します
- 「--image」、「--memwidth」、「--romwidth」(メモリ幅 / ROM 幅の設定)。この幅は、CPU1 / CPU2 では 16 に、CM では 8 に設定します。
- 対応するコア用のフラッシュ HEX リンカ コマンド ファイルのパス。


Copyright © 2026 Texas Instruments Incorporated

## 7 16KB を超えるフラッシュ コードの認証

セキュア フラッシュ ブートは、エントリ アドレスからフラッシュ セクタの最初の 16KB のみを認証します。フラッシュの他のセクタを認証するには、ユーザー アプリケーションでセキュア フラッシュ ブートの CMAC API を直接呼び出す必要があります。

hex ユーティリティを使用すると、4 つの各フラッシュ エントリ アドレス +16KB、および 1 つのカスタム フラッシュ範囲に対するゴールデン CMAC タグの生成がサポートされます。カスタム フラッシュ範囲は、カスタム アドレス範囲に対して CMAC 認証を実行できるように構成できます。これは、必要に応じてすべてのフラッシュ セクタの長さにすることもできます。

カスタム フラッシュ範囲認証用の **CPU1 / CPU2** アプリケーション **CMAC** 構造

```
struct CMAC_TAG
{
   char tag[8];
   uint32_t start;
   uint32_t end;
}
```

全フラッシュ範囲認証用の **CPU1 / CPU2** ゴールデン **CMAC** タグのメモリ アロケーション

```
#pragma RETAIN(cmac_all)
#pragma LOCATION (cmac_all, 0x087002)
const struct OMAC_TAG cmac_all = {{0}, 0x0, 0x0};
```

カスタム フラッシュ範囲認証用の **CM** アプリケーション **CMAC** 構造体

```
struct CMAC_TAG
{
   uint8_t tag[16];
   uint32_t start;
   uint32_t end;
}
```

全フラッシュ範囲認証用の **CM** ゴールデン **CMAC** タグのメモリ アロケーション

```
#pragma RETAIN(cmac_all)
#pragma LOCATION (cmac_all, 0x00204004)
const struct OMAC_TAG cmac_all = {{0}, 0x0, 0x0};
```

カスタム CMAC 認証範囲を作成するために、「cmac_all」という構造体を作成します。

- LOCATION プラグマを使用して、CMAC ゴールデン タグ用に、認証対象範囲内の任意のアドレスを指定します。(CMAC ゴールデン タグが認証開始アドレスに近いほど、CMAC アルゴリズムの実行は高速になります)。
- 「tag」構造体要素は常にゼロで初期化したままにします。
- 「start」および「end」構造体要素を初期化して、カスタム範囲を設定します。両方が 0 の場合、プライマリ セキュア ブートの 16KB メモリ範囲を含め、デバイス コア フラッシュ メモリ全体が認証されます。
- アプリケーションの CMAC 変数 / 構造体に関する追加の詳細については、[3] および [4] を参照してください。

**CPU1 セキュア フラッシュ CMAC 認証 API**

```
applicationCMACStatus = CPU1BROM_calculateCMAC (CMAC_AUTH_START_ADDRESS,
                                                CMAC_AUTH_END_ADDRESS,
                                                CMAC_AUTH_TAG_ADDRESS);
```

アプリケーションでは、F2838x セキュア ゾーン コード シンボル ライブラリ (C2000Ware [2] の <C2000Ware_Install_Directory>/libraries/boot_rom/f2838x にある) をプロジェクトに追加し、該当するコアのセキュア フラッシュ ブート CMAC API を呼び出します。CPU1 での API 呼び出し例を、上の例に示します。

- CMAC_AUTH_TAG_ADDRESS は、LOCATION プラグマで指定したものと一致している必要があります


Copyright © 2026 Texas Instruments Incorporated

- CMAC_AUTH_START_ADDRESS と CMAC_AUTH_END_ADDRESS は、「cmac_all」構造体で指定したものと一致している必要があります。また重要な注意点として、開始アドレスと終了アドレスを 128 ビットに揃える必要があります。
- たとえば、F2838x の CPU1 フラッシュ メモリ全体を認証するための開始アドレスと終了アドレスは、次のようになります。
  - 開始:0x00080000
  - 終了:0x000C0000
- セキュア フラッシュ ブートの CMAC API から返されるステータスは、ユーザー アプリケーションで確認し、適切に処理する必要があります。
- セキュア フラッシュ CMAC 認証 API の詳細については、[1] の「ROM コードおよびペリフェラル ブート」章を参照してください。

注

上記の例では実際の終了アドレスは 0x000BFFFF ですが、終了アドレスを 128 ビットに揃えるために、0x000C0000 として指定する必要があります。

## 8 デバッグ リソース

**表 8-1. セキュア フラッシュ ブートのデバッグ シナリオ**

| シナリオ | 動作 |
|---|---|
| CPU1 でセキュア ブートに失敗 | スタンドアロン ブート:デバイスがリセットされました。<br>エミュレーション ブート:CPU1 はアドレス範囲 0x3FB13C ～ 0x3FB142 内で停止します |
| CPU2 でセキュア ブートに失敗 | CPU2TOCPU1IPCBOOTSTS (1) レジスタのビット 21 がセットされます。<br>CPU2 はセキュア フラッシュ CMAC エラー コードを伴う IPC コマンドを CPU1 に送信します。 |
| CM でセキュアブートに失敗 | CMTOCPU1IPCBOOTSTS (2) レジスタのビット 21 がセットされます。<br>CM はセキュア フラッシュ CMAC エラー コードを伴う IPC コマンドを CPU1 に送信します。 |
| CPU1 のセキュア ブート実行が成功しているのか? | 0x0000 0002 アドレスにある CPU1 BootROM ステータスのビット 7:0 に 0x3 が反映されます。 |
| CPU2 のセキュア ブートの実行が成功しているのか? | CPU2TOCPU1IPCBOOTSTS (1) レジスタのビット 7:0 に 0x3 が反映されます。 |
| CM のセキュア ブート実行が成功しているのか? | CMTOCPU1IPCBOOTSTS (2) レジスタのビット 7:0 に 0x3 が反映されます。 |

1. 同じ情報は、CPU2 のアドレス 0x0000 0002 の位置にもキャプチャされます。
2. 同じ情報は、CM のアドレス 0x2000 0000 の位置にもキャプチャされます。

## 9 追加情報および考慮事項

- 推奨しませんが、セキュア フラッシュ ブートを CPU1 ではなく CPU2/CM で実行する場合は、CPU2/CM をリセットから解除する前に、Z1 OTP CMACKEY に対して CPU1 からダミー ロードを実行する必要があります。ダミー ロードは、CPU1 USER OTP の 0x78018 ～ 0x7801F 領域を読み取ることで実行されます。
- 同様に、CPU1 でセキュア フラッシュ ブート モードを実行せずに CMAC 認証 API を使用する場合は、API を呼び出す前に、Z1 OTP CMACKEY に対して CPU1 からダミー ロードを実行する必要があります。
- CPU1 でセキュア フラッシュ ブートを使用する際は、BOOTDEF テーブルにおいて、通常 (非セキュア) のフラッシュ ブート モードを、セキュア フラッシュ ブートと同じセクタに設定しないことを推奨します。
- 16KB を超えるフラッシュ コードを認証する場合:
  - 128 ビットのゴールデン CMAC タグは、計算が実行されるメモリ アドレス範囲内に格納する必要があります。
  - ゴールデン CMAC タグの開始アドレスは、32 ビット境界に整列させる必要があります。


Copyright © 2026 Texas Instruments Incorporated

- ブート モード設定はワン タイム プログラマブル (OTP) 内にあるため、ブート関連の構成を確定する前に、エミュレーション ブート モードを使用して試行することを推奨します。エミュレーション ブートの詳細については、『TMS320F2838x テクニカル リファレンス マニュアル』の「ROM コードおよびペリフェラル ブート」の章を参照してください [1]。


Copyright © 2026 Texas Instruments Incorporated

# 10 C2000 CMAC アルゴリズムの OpenSSL への整合

C2000™ hex ユーティリティ (hex2000) は、本ドキュメント全体を通し「スワップワード」と呼ばれる独自のバイト再順序変換を適用します。この変換は、CMAC 計算を行う前のメッセージ データに適用されるほか、生成された認証タグについてもフラッシュ イメージに組込む前に適用されます。これは、セクション 10.2 で詳細に説明しています。

OpenSSL の AES-128-CMAC 実装は、このような順序変換を行わない、変更のない連続したバイト ストリームに対して動作するため、raw バイナリで「openssl dgst -mac」cmac コマンドを直接呼び出しても、C2000 で生成されるゴールデン CMAC タグに一致する出力は生成されません。このセクションでは、OpenSSL の出力を C2000 と整合させるために必要な正確な動作シーケンスについて説明します。これにより開発者は、標準のオープンソース暗号ツールを用いて、ゴールデン CMAC タグを独立して検証または再現できるようになります。

この整合化手順は、開発および検証ワークフローで使用することを目的としています。生産向けのイメージ生成における hex2000 ユーティリティの代替ではありません。「hex2000 --cmac」フラグとキー ファイル形式の詳細については、[3] および [4] を参照してください。

## 10.1 C28x メモリおよびバイナリ ファイルのバイト順序

C28x CPU は、16 ビット ワードアドレス方式です (バイト単位でアドレス指定できません)。各 C28x ワードは 16 ビット幅です。バイナリ ファイルでは、各 16 ビット ワードはリトル エンディアンで格納されます (下位バイトがより低いファイル オフセットに配置されます)。

### バイナリ ファイルでの 32 ビット値の見え方

C28x の uint32_t 値 0xAABBCCDD がワード アドレス N に格納される場合:

```
C28x word address N:   value = 0xCCDD  (low 16 bits)
C28x word address N+1: value = 0xAABB  (high 16 bits)

Binary file bytes:
  offset 2k+0: 0xDD  ← low  byte of word[N]
  offset 2k+1: 0xCC  ← high byte of word[N]
  offset 2k+2: 0xBB  ← low  byte of word[N+1]
  offset 2k+3: 0xAA  ← high byte of word[N+1]
```

したがって、uint32_t 0xAABBCCDD はバイナリ ファイルでは次のように表示されます: DD CC BB AA

### 例:コードスタート分岐命令

C28x 0x080000 の「LB 0x081BC8」命令:

```
C28x word 0x080000: 0x4800  → binary bytes: 00 48
C28x word 0x080001: 0x1BC8  → binary bytes: C8 1B

Binary file at offset 0: 00 48 C8 1B
```

## 10.2 フラッシュ バイナリのバイト順序

hex2000 ユーティリティは、フラッシュ バイナリを AES-128-CMAC アルゴリズムに直接供給しません。認証タグを計算する前に、バイナリ データの各 32 ビット グループ内にある 2 つの 16 ビット ワードを入れ替える「スワップワード」変換を適用します。

バイナリ内の連続する 4 バイトの各グループ [b0, b1, b2, b3] について、ここで、

- [b0, b1] = アドレス N にある C28x ワードの [下位バイト、上位バイト]
- [b2, b3] = アドレス N+1 にある C28x ワードの [下位バイト、上位バイト]

```
swapwords([b0, b1, b2, b3]) = [b2, b3, b0, b1]
```

これは、各 32 ビット グループ内の 2 つの 16 ビット ワードを入れ替えますが、各ワードのバイトは元の (リトル エンディアンの) 順序に保たれます。これは、各 C28x ワードのペアを逆の順序で読み取ること (N の下位ワードより前に N+1 の上


Copyright © 2026 Texas Instruments Incorporated

位ワードを読み取ること) に相当します。この操作は自己可逆であり、どのような入力に対しても、この変換を 2 回適用すると元のデータが変化せずに返されます。

hex2000 は、AES-128-CMAC タグを変換後のデータに対して計算した後、その 16 バイトのタグ自体に同じスワップワード変換を再度適用します。その後、このタグをフラッシュ イメージ内のゴールデン CMAC タグの位置 (フラッシュ バンク エントリ アドレスから +0x4 バイト オフセット、C28x ワード アドレスの +0x2 に相当) に組込みます。OpenSSL を使用する場合、hex2000 で生成されたものとバイト単位で完全に一致する保存タグを生成するには、これらの変換の両方を複製する必要があります。

## 10.3 CMAC キーのバイト順序

128 ビットの CMAC キーは、DCSM ゾーン 1 の OTP に 4 つの 32 ビットレジスタ (DCSM_Z1_CMACKEY0 ～ DCSM_Z1_CMACKEY3) としてプロビジョニングされています。ここで、CMACKEY0 が最上位 32 ビットを、CMACKEY3 が最小桁 32 ビットを保持します。

hex2000 のキー テキスト ファイル形式では、これら 4 つの値を最上位から最小桁へと順に連結し、接頭辞「0x」を付与します。OpenSSL の「-macopt hexkey:<32-hex-key>」オプションは、同じ 32 文字の 16 進数文字列を、追加のバイト順序の変更なしにそのまま受け付けます。表 10-1 に、OTP レジスタ値、hex2000 のキー ファイル形式、および OpenSSL hexkey 引数の対応を示します。

**表 10-1. CMAC キーのバイト順序の対応**

| OTP レジスタ | 値 (例) | 職種 |
|---|---|---|
| DCSM_Z1_CMACKEY0 | 0x2b7e1516 | キー ビット [127:96] (MSW) |
| DCSM_Z1_CMACKEY1 | 0x28aed2a6 | キー ビット [95:64] |
| DCSM_Z1_CMACKEY2 | 0xabf71588 | キー ビット [63:32] |
| DCSM_Z1_CMACKEY3 | 0x09cf4f3c | キー ビット [31:0] (LSW) |
| **hex2000 キー ファイル** | 0x2b7e151628aed2a6abf7158809cf4f3c | 単一行、0x 接頭辞付き |
| **OpenSSL の hexkey** | 2b7e151628aed2a6abf7158809cf4f3c | 同じ値、0x 接頭辞なし |

## 10.4 CMAC 出力の整列手順

OpenSSL を使用して hex2000 ゴールデン CMAC タグを再現するプロセスは、5 つのステップから成ります。各ステップは次のとおりです。

ステップ **1 —** 署名前のバイナリを取得する。

CMAC 署名前の、16KB (0x4000 バイト) のフラッシュ バイナリから開始します。この状態では、バイト オフセット 0x4 にある 16 バイトのゴールデン CMAC タグのプレースホルダ (cmac_sb_1) は、セクション 6 で説明されているアプリケーション ソース コードの要件に従って、すべて 0 に初期化する必要があります。16KB 領域の残りの部分は、意図したフラッシュの内容を反映しており、プログラムされていない領域は 0xFF で埋められている必要があります。

ステップ **2 —** ゴールデン **CMAC** タグ領域をマスクする。

CMAC ゴールデン署名は、CMAC がカバーする 16KB の領域に格納されます。循環依存 (タグが自身に依存すること) を回避するため、CMAC 計算中に署名領域はすべて 1 (0xFFFF) でマスクします。C28x デバイスでは、消去されたフラッシュはすべて 1 として読み取られます。署名領域をマスキングすることで、CMAC 計算ではその署名位置を消去済みフラッシュのように扱います。これは、プログラミング前のフラッシュの初期状態と一致しています。

したがって、CMAC を計算する前に、バイト 0x4 ～ 0x13 (両端を含む) を値 0xFF に置き換えます。このマスキング ステップは、計算に hex2000 か OpenSSL のどちらを使用するかに関係なく実行する必要があり、セクション 5 で説明されている動作と整合しています。

ステップ **3 —** スワップワード変換を適用して **OpenSSL** 入力を生成する。

16KB のマスク済みバイナリ全体に対し、スワップワードのバイト並び替えを適用します。連続した 4 バイトの各グループ [b0、b1、b2、b3] について、その場所に [b2、b3、b0、b1] を書き込みます。結果として得られた 16KB のバイト シーケン


Copyright © 2026 Texas Instruments Incorporated

スを、一時ファイル (例:input_swaped.bin) に書き込みます。この変換されたファイルが、OpenSSL の CMAC コマンドへの直接入力となります。

**ステップ 4 — OpenSSL を使用して AES-128-CMAC を計算する。**

適切な 32 文字の 16 進数キー文字列に置き換え、次のように OpenSSL を呼び出します。

**変換済み入力に対して AES-128-CMAC を計算するための OpenSSL コマンド**

```
openssl dgst -mac cmac \
             -macopt cipher:AES-128-CBC \
             -macopt hexkey:<32-hex-key> \
             -binary input_swapped.bin > aes_tag.bin
```

出力ファイル aes_tag.bin には、AES-128-CMAC の生のタグを表す 16 バイトがそのまま格納されます。この値は、hex2000 が最終変換を適用する前に計算する内部タグに相当します。

**ステップ 5 — タグにスワップワードを適用し、格納形式を取得する。**

ステップ 4 で作成した 16 バイト AES タグに、同じスワップワード変換を適用します。結果として得られる 16 バイトは、hex2000 が使用し、ブート ROM が期待する格納形式のゴールデン CMAC タグとなります。この値は、署名付きフラッシュ イメージのバイト オフセット 0x4 に hex2000 が書き込むバイトと一致している必要があります。またこの値は、認証中にブート ROM がゴールデン CMAC タグの位置 0x00080002 (C28x ワードアドレス) から読み取る値です。

スワップワードは自己可逆であるため、ブート ROM は格納されたタグを読み取り、スワップワードをもう一度適用するだけで元の AES タグを復元できます。その後、マスク済み / 変換済みフラッシュ内容に対して再計算したタグと比較します。

## 10.5 動作例

次の例では、NIST AES-128 テスト キー 2b7e151628aed2a6abf7158809cf4f3c を最小構成のフラッシュ イメージに適用しています。最初のアプリケーション ワード (オフセット 0x0) は、2 つの C28x ワード「0x4800」および「0x1BC8」としてアセンブルされた「LB 0x081BC8」分岐命令です。オフセット 0x4 のゴールデン CMAC タグのプレースホルダはゼロに初期化されます。オフセット 0x14 以降のフィールドは、消去状態 (0xFF) になっています。残りの 16KB 領域は、0xFF で埋められています。

**署名前のバイナリ (最初の 24 バイトを表示、残りは 0xFF)**:

```
offset 0x0000:  00 48 C8 1B   (codestart: LB 0x081BC8)
offset 0x0004:  00 00 00 00   (cmac_sb_1, initialized to zero)
offset 0x0008:  00 00 00 00   (cmac_sb_1, continued)
offset 0x000C:  00 00 00 00   (cmac_sb_1, continued)
offset 0x0010:  00 00 00 00   (cmac_sb_1, continued)
offset 0x0014:  FF FF FF FF   (erased contents)
```

ステップ **2** 後 **— 署名領域は 0xFF** でマスクされた状態:

```
offset 0x0000:  00 48 C8 1B   (unchanged)
offset 0x0004:  FF FF FF FF   (masked)
offset 0x0008:  FF FF FF FF   (masked)
offset 0x000C:  FF FF FF FF   (masked)
offset 0x0010:  FF FF FF FF   (masked)
offset 0x0014:  FF FF FF FF   (unchanged)
```

ステップ **3** 後 **—** スワップワードが適用された状態 **(**最初の **8** バイトを表示**)**:

```
offset 0x0000:  C8 1B 00 48   ([b2,b3,b0,b1] of original offset 0x0000)
offset 0x0004:  FF FF FF FF   (unchanged; all bytes identical)
```

ステップ **4 — OpenSSL** の **AES-128-CMAC** 出力 **(aes_tag.bin**、**16** バイト**)**:

```
f2 70 99 41  e1 cf 3c cf  f1 36 7a a3  c4 3e f6 f7
```


Copyright © 2026 Texas Instruments Incorporated

ステップ **5** 後 **—** タグにスワップワードが適用された状態 **(**格納形式**)**:

```
99 41 f2 70  3c cf e1 cf  7a a3 f1 36  f6 f7 c4 3e
```

最終署名済みバイナリ **(**最初の **32** バイト**)**:

```
Offset 0x0000:  00 48 C8 1B
Offset 0x0004:  99 41 F2 70    (stored tag bytes 0-3)
Offset 0x0008:  3C CF E1 CF    (stored tag bytes 4-7)
Offset 0x000C:  7A A3 F1 36    (stored tag bytes 8-11)
Offset 0x0010:  F6 F7 C4 3E    (stored tag bytes 12-15)
Offset 0x0014:  FF FF FF FF
```

上記の格納されたタグは、同じキーを使用して同じバイナリに適用した「hex2000 --cmac」の出力と一致します。この結果は、hex2000 出力とバイト単位で一致していることが確認されています。

### 10.6 相違点のまとめ

「hex2000 --cmac」署名フローと、OpenSSL AES-128-CMAC コマンドをそのまま呼び出した場合における動作上の唯一の違いは、AES 動作の前後で行われる 2 回のスワップワード変換です。どちらにおいても、基盤となる暗号アルゴリズム、鍵スケジュール、初期化ベクトルは同じです。表 10-2 では、参考として 2 つのフロー間の完全な対応関係をまとめています。

**表 10-2. C2000 と OpenSSL の CMAC フローの比較**

| 要素 | C2000 | OpenSSL |
|---|---|---|
| メッセージの前処理 | AES の前にスワップワード (masked_binary) を適用 | 生のバイト ストリーム (変換の適用なし) |
| 署名領域のマスキング | バイト オフセット 0x4 ～ 0x13 を 0xFF で置き換え | OpenSSL を呼び出す前に手動で適用する必要あり |
| キー バイト順序 | MSB ファースト (CMACKEY0 \|\| ... \|\| CMACKEY3) | MSB ファースト。同じ順序で、変換は不要 |
| AES-128-CMAC アルゴリズム | NIST SP 800-38B (OMAC1) | NIST SP 800-38B (OMAC1)、同一 |
| 初期化ベクトル | すべてゼロ (CMAC 標準で暗黙的に規定) | すべてゼロ (CMAC 標準で暗黙的に規定) |
| タグの後処理 | 組込み前にスワップワード (aes_tag) を適用 | 生の AES タグ出力。スワップワードは手動で適用する必要あり |
| 出力 | 保存タグが組込まれた署名済みバイナリ | 16 バイトの生の AES タグ。追加の処理が必要 |

---

注

スワップワード変換は 32 ビット グループで動作するため、認証対象領域の長さは 4 バイトの倍数である必要があります。16KB (0x4000 バイト) の C28x プライマリ セキュア ブート領域は、この要件を満たしています。セクション 7 に示すように、カスタム フラッシュ範囲に対して拡張認証を実行する場合、CMAC アルゴリズムがパディングなしで正しく動作するためには、開始アドレスと終了アドレスを同様に 128 ビットに揃える必要があります。

---

## 11 参考資料

1. テキサス・インスツルメンツ:*TMS320F2838x リアルタイム マイコン テクニカル リファレンス マニュアル*
2. C2000 リアルタイム MCU 向け C2000Ware
3. テキサス・インスツルメンツ:*TMS320C28x アセンブリ言語ツール v20.2.0.LTS ユーザー ガイド*
4. テキサス・インスツルメンツ:*Arm アセンブリ言語ツール v20.2.0.LTS ユーザー ガイド*
5. TMS320F28338D 製品ページ


Copyright © 2026 Texas Instruments Incorporated

## 12 改訂履歴

**Changes from Revision * (September 2020) to Revision A (June 2026)** **Page**


- 新しい C2000 デバイスにおけるセキュア ブートのサポートを更新。 ... 1
- ドキュメント全体にわたって表、図、相互参照の採番方法を更新 ... 2
- C2000 実装の OpenSSL への整合の詳細を示すセクションを追加。 ... 10



Copyright © 2026 Texas Instruments Incorporated

# 重要なお知らせと免責事項

TI は、技術データと信頼性データ (データシートを含みます)、設計リソース (リファレンス デザインを含みます)、アプリケーションや設計に関する各種アドバイス、Web ツール、安全性情報、その他のリソースを、欠陥が存在する可能性のある「現状のまま」提供しており、商品性および特定目的に対する適合性の黙示保証、第三者の知的財産権の非侵害保証を含むいかなる保証も、明示的または黙示的にかかわらず拒否します。

これらのリソースは、TI 製品を使用する設計の経験を積んだ開発者への提供を意図したものです。(1) お客様のアプリケーションに適した TI 製品の選定、(2) お客様のアプリケーションの設計、検証、試験、(3) お客様のアプリケーションに該当する各種規格や、その他のあらゆる安全性、セキュリティ、規制、または他の要件への確実な適合に関する責任を、お客様のみが単独で負うものとします。

上記の各種リソースは、予告なく変更される可能性があります。これらのリソースは、リソースで説明されている TI 製品を使用するアプリケーションの開発の目的でのみ、TI はその使用をお客様に許諾します。これらのリソースに関して、他の目的で複製することや掲載することは禁止されています。TI や第三者の知的財産権のライセンスが付与されている訳ではありません。お客様は、これらのリソースを自身で使用した結果発生するあらゆる申し立て、損害、費用、損失、責任について、TI およびその代理人を完全に補償するものとし、TI は一切の責任を拒否します。

TI の製品は、 TI の販売条件 、 TI の総合的な品質ガイドライン 、 ti.com または TI 製品などに関連して提供される他の適用条件に従い提供されます。TI がこれらのリソースを提供することは、適用される TI の保証または他の保証の放棄の拡大や変更を意味するものではありません。 TI がカスタム、またはカスタマー仕様として明示的に指定していない限り、TI の製品は標準的なカタログに掲載される汎用機器です。

お客様がいかなる追加条項または代替条項を提案する場合も、TI はそれらに異議を唱え、拒否します。

Copyright © 2026, Texas Instruments Incorporated

最終更新日：2025 年 10 月