参考資料

Application Note

F28P65x デバイスでのデュアルコア デバッグ

Ira Thete, Mithra Kandasamy

概要

C28x ファミリのマイコンは、多くのデュアルコア デバイスで構成されています。これらは、同一のコアである場合もあれば、異なるコアである場合もあります。この種のデバイスの主要なアプリケーション分野は、産業用ドライブ、ソーラー インバータ、パワー ストレージなどです。各コアは、アプリケーション内でさまざまな機能を担います。たとえば、産業用ドライブの場合、1 番目のコアを使用して内部の電流 / 速度制御ループを処理し、2 番目のコアを使用して I/O 通信や故障処理の調整を行うことができます。このアプリケーション ノートは、デュアルコア C28x デバイスのプログラムとデバッグを行う方法と、避けるべき一般的なミスについて解説しています。

目次

商標

1 概要

リアルタイム制御アプリケーションに関しては、常に根本的な懸念事項が存在します。決定論的なハードリアルタイム制御ループ (通常は 100kHz 程度の周波数で動作) では、通信、保護ロジック、テレメトリ、故障管理などの低速の監視タスクと演算時間を共有できません。2 つ目の CPU を導入することで、全体的なワークロードを分割し、リアルタイム制御関連のタスクに単一の CPU の専用帯域幅を使用できるため、この問題点が緩和できます。このアプリケーション ノートでは、デュアルコア環境で予期しない動作をする可能性のあるアプリケーションを効果的にデバッグする方法について説明します。ブート シーケンスやメモリ / ペリフェラルの所有権の競合で、デュアルコア アプリケーションのパフォーマンスが不安定になる原因についても解説します。

2 C28x ファミリのデュアル コア アーキテクチャ

C2000 デュアルコア製品群は、以下の 2 つのアーキテクチャ構成に分類されます。

2.1 同種アーキテクチャ (C28x + C28x)

どちらの CPU も、同一の C28x DSP コアで、同じ ISA、同じコンパイラ ツールチェーン、および FPU や TMU などの同じ命令セット拡張を実行します。2 つのコアは機能的に対称で、どちらも任意のタスクを実行できます。そのようなデバイスの例は次のようなものです

- TMS320F2837x
- TMS320F28P65xx

ハードウェアが対称型であるものの、スタートアップ時にコアは同じようには機能しません。CPU1 はシステム マスタ CPU としてハードワイヤードされています。

2.2 異種混在アーキテクチャ (C28x + C28x + ARM Cortex M4)

この構成では、デュアル コアの C28x サブシステムは、従来のようにリアルタイム制御を担います。ARM Cortex M4「コネクティビティ マネージャ」は、125MHz で動作する汎用 32 ビット RISC コアで、通信ミドルウェアと RTOS ベースのタスクを実行するための専用設計です。

2.2.1 TMS320F2838x (F28388D、F28386D など)

F2838xD ファミリのデバイスには、2 つの C28x CPU サブシステムと 1 つの CM (コネクティビティ マネージャ) サブシステムが含まれています。各 C28x CPU サブシステムには次のものが含まれます

1. 1 つの C28x CPU
2. 1 つの CLA (制御補償器アクセラレータ) コプロセッサ
3. 1 つの DMA (ダイレクト メモリ アクセス ユニット)

C28x1 および C28x2 コアは、それぞれ CPU1 および CPU2 とも呼ばれることに注意してください。マルチコア デバッグの重要な点のひとつは、どの CPU がどのハードウェア ペリフェラルの所有権を持つかを決めることです。F2838xD CPU には、複数のメモリ ブロック、ペリフェラル、GPIO が搭載されています。これらの中には、特定のコアのみが所有できるものもあれば、複数のコア間で共有できるものもあります。「複数のコアで共有可能」というカテゴリの下にあるすべてのリソースは、いかなる機能を実行する前に、明示的な所有者を割り当てる必要があります。

3 デュアルコア アプリケーションの一般的なフロー

- マルチコア デバッグの重要なルールとして、この所有権が常に CPU1 によって決定されるという点があります。
- もう 1 つの重要な側面は、2 つの CPU 間で効果的な通信とデータ共有を可能にすることです。これは、プロセッサ間通信モジュールにより処理されます。これに加えて、本デバイスには追加の専用メッセージ RAM があり、データの共有に使用できます。
- 両方のコアはハードウェアが同一であるにもかかわらず、マルチコア システムにおいては自律性のレベルが異なります。CPU1 はマスタ コアとして動作します。クロック設定やすべての GPIO の初期化を含むデバイスの基本的な初期化は、CPU1 のアプリケーションによって実行されます。CPU1 は、BOOT ピンの構成に基づいて、さまざまなモードでブートできます。CPU2 および CM (F2838Xd の場合) コアは、IPC モジュールを使用して CPU1 アプリケーションでブートする必要があります。

4 ブート シーケンス

デバイスがリセット状態から復帰すると、CPU1 のブート ROM のみが実行されます。CPU2 はハードウェアによってリセット状態に保持され、CPU1 が明示的に解除するまでは、いかなるコードも実行できません。これは、以下のことを意味しています。

- **システム クロックと PLL** は、CPU1 によってのみ初期化されます。CPU1 が PLL を設定し、ロックするまで待機する間、デバイスは内部発振器を使い、速度を落として動作します。CPU2 は、CPU1 が設定したクロック状態をそのまま引き継ぎます。
- **ペリフェラルの所有権**は、CPU1 によって割り当てられます。このデバイスには、いずれのコアにも割り当て可能な一連のペリフェラルが搭載されていますが、割り当てレジスタは CPU2 からの書き込みが保護されています。どのコアがどのペリフェラルを所有するかを決定できるのは、CPU1 のみです。
- **GPIO の初期化**は、CPU1 によって実行されます。CPU2 が GPIO を安全に使用できるようになるためには、すべての GPIO のマルチプレクサ、方向、およびプルアップの設定を、事前に CPU1 で設定しておく必要があります。
- システム全体の**ウォッチドッグ**は CPU1 によって管理されます。CPU2 には独自のウォッチドッグがありますが、システム レベルの保護は CPU1 で開始されます。

リセットのたびに、次のシーケンスが発生します。

1. **CPU1 のブート ROM が実行開始。**ブート ROM は、ブート モード ピン (デバイスに応じて GPIO84、GPIO72 など) を読み取り、JTAG、SCI、SPI、CAN、I2C、パラレル、またはフラッシュから直接などのブート方法を決定します。これはブートローダー フェーズであり、すべて CPU1 で実行されます。
2. **CPU1 アプリケーションが起動。**ブートローダーが制御を引き渡すと、CPU1 はアプリケーションの実行を開始します。適切に記述された CPU1 アプリケーションが最初に行うことは、次のとおりです。
   - – デバイスを初期化 (PLL、クロック、フラッシュの待機状態)
   - – ペリフェラルとメモリの所有権割り当て
   - – CPU2 が必要とする共有メモリ領域の初期化
3. **CPU1 が CPU2 をリセット状態から解除。**CPU2 は単独で起動できません。CPU1 は、CPU2 のブート シーケンスを呼び出す必要があります。driverlib においては、これは Device_bootCPU2() です。この関数は、IPC レジスタに書き込むことで、CPU2 のブート ROM に実行を通知し、CPU2 のブート モードを指定します (通常はフラッシュからブートするか、CPU1 によってロードされた特定の RAM アドレスからブートします)。
4. **CPU2 のブート ROM が実行開始。**CPU2 のブート ROM は、CPU1 が IPC 経由で指定したブート モードを使用して、自らのブート シーケンスを実行します。CPU1 が CPU2 にフラッシュからブートするよう指示した場合、CPU2 はフラッシュ エントリ ポイントにジャンプします。CPU1 が CPU2 を SARAM からブートするように指示した場合 (CPU1 がすでに CPU2 のイメージをコピーした RAM デバッグ ビルドでは有用)、CPU2 はロードされたコードにジャンプします。
5. **両方のコアが独立して動作。**この時点以降、両方の CPU がそれぞれのアプリケーションを同時に実行します。両者の間の連携は、IPC フラグと共有 RAM / メッセージ RAM によって完全に処理されており、通常の実行中は両者の間にこれ以上のハードウェア依存関係は存在しません。

5 単純なデュアル コアのサンプルを実行する

デュアルコア環境に関連する概念を理解するための第一歩としては、デュアルコア c28x デバイスで LED を点滅させる簡単なサンプル プログラムを実行するのが良いでしょう。このサンプルでは、F28P65x LaunchPad を使用します。このサンプルを実行するには、最近のバージョンの c2000ware と Code Composer Studio が必要です。

CPU1 と CPU2 で LED を点滅させる方法

ステップ 1 — 両方のプロジェクトをインポートする

C:/ti/C2000Ware_x_x_x_x/driverlib/f28p65x/examples/c28x_dual/led/CCS/ に移動し、「select folder」(フォルダを選択) をクリックします。2 つの異なるプロジェクトが表示されます。**led_ex1_c28x_dual_blinky.projectspec** を必ずインポートしてください。これにより、CPU1 と CPU2 の両方のプロジェクトが CCS ワークスペースにインポートされます。

ステップ 2 — ビルド構成を選択する

RAM 構成でデバッグする場合

1. 両方のプロジェクトのビルド構成を CPUx_LAUNCHXL_RAM に変更します
2. 最初に CPU1 プロジェクトをビルドし、次に CPU2 をビルドします。どちらも正常にビルドされ、led_ex1_c28x_dual_blinky_cpu1\CPU1_LAUNCHXL_RAM\led_ex1_c28x_dual_blinky_cpu1.out と led_ex1_c28x_dual_blinky_cpu2\CPU2_LAUNCHXL_RAM\led_ex1_c28x_dual_blinky_cpu2.out がそれぞれ生成されます。
3. ターゲット構成ファイル (.ccxml ファイル) を右クリックし、「start a project-less debug」(プロジェクトレス デバッグの開始) をクリックします。
4. この作業を完了すると CCS デバッグ セッションが開始されますが、どのコアにもプログラムはロードされません。デバッグ ビューの「Thread」(スレッド) セクションに、使用可能なコアがすべて表示されます。
5. 「THREADS」(スレッド) セクションで C28xx_CPU1 と C28xx_CPU2 を (この順序で) 右クリックし、「Connect Target」(ターゲットを接続) オプションを選択して、これらを接続します。
6. 両方のコアが接続されたら、「C28xx_CPU1」を選択し、「Run」(実行) →「Load」(ロード) →「Load Program」(プログラムをロード) の順に移動し、「led_ex1_c28x_dual_blinky_cpu1.out」まで移動して選択します
7. これにより、CPU1 のプログラム実行が main() で停止していることがわかります。プログラムを実行すると、Launchpad で LED4 の点滅を確認できます。
8. CPU2 のコードを実行する前に、CPU1 コードが Device_bootCPU2() 関数の実行を終了していることが非常に重要です。
9. CPU1 コードを実行したまま (または、Device_bootCPU2() 関数実行後のポイントで停止させて)、led_ex1_c28x_dual_blinky_cpu2.out イメージを C28xx_CPU2 にロードできます。
10. すべての手順が正しく実行されていれば、プログラムは CPU2 でも main() で停止し、正常に実行できます。Launchpad で LED5 の点滅を確認できます。

フラッシュ構成でデバッグする場合

1. 両方のプロジェクトのビルド構成を CPUx_LAUNCHXL_FLASH に変更します
2. CPU1 のターゲット構成ファイル (.ccxml) で、プロジェクトレス デバッグを開始します。
3. ターゲットを CPU1 と CPU2 スレッドに接続します。
4. CPU1 のプロパティに移動し、ドロップダウンで「flash bank flash settings」(フラッシュ バンクのフラッシュ設定) を開きます。「Flash Bank Map Settings」(フラッシュ バンクのマップ設定) まで下にスクロールして、バンク 3 とバンク 4 に 1 を設定します (0 は CPU1、1 は CPU2)。
5. 「configure clock」(クロックの設定) をクリックします。
6. 下にスクロールして「erase settings」(設定消去) に移動し、「Selected banks only」(選択したバンクのみ) にチェックを入れてから、バンク 0、1、2 を選択します (これらは CPU1 用です)。
7. 「CPU2 properties」(CPU2 のプロパティ) をクリックします。「Flash Bank Map Settings」(フラッシュ バンクのマップ設定) が有効になっている場合、バンク 3 とバンク 4 に再度 1 を設定します。無効化されている場合は、そのままにします。

8. 再度、「erase settings」(設定消去) まで下にスクロールし、バンク 3 と 4 を選択します。デバッグ コンソールに「Device is in reset」(デバイスがリセット状態です) と表示されている場合は、USB を取り外して再度接続するか、新しいセッションを開きます。それでもエラーが発生する場合でも、プログラムは関係なく実行されます。
9. ここで、「CPU1 thread」(CPU1 スレッド) をクリックして、「Run」(実行) →「Load」(ロード) -→「Load program」(プログラムをロード) の順に移動します。
10. 次に、(以前のビルドで生成された) それぞれの .out ファイルを参照し、プログラムを実行します。(注:CPU1 プログラムが、少なくとも関数 Device_bootCPU2() まで実行されることを確認してください。)CPU2 スレッドでも同じ手順に従います。

サンプル実行中によく発生する問題

hex2000 ユーティリティが認識されない

特に Windows PowerShell を使用している場合は、元のプロジェクト パスを使用して参照しなければならないことがあります。hex2000 だけを記述する代わりに、hex2000 のフル パスを記述します。

combined.hex を CPU1 にロードした後、メモリが正しいアドレスに更新されない

ブート ROM のシーケンスは状況によって異なる場合があり、CPU2 がまだデフォルトのブート アドレスに留まっている可能性があります。逆アセンブリ ビューを確認し、ポインタの位置を手動で変更して、メモリに何が格納されているかを確認してください。メモリが更新された場合は、Launchpad を切り離し (リセットをトリガします)、再度接続して確認します。メモリ内での更新がされていない場合は、combined.map ファイルをチェックして、結合された出力ファイルが正しくビルドされているかどうかを確認します。そうでない場合は、プロジェクトを新規にインポートして、もう一度やり直してください。

両方の CPU が実行されているが、LED が点滅しない

プログラムが停止している場所を調べるには、ブレークポイントを設定します。IPCsync で停止した場合は、プロジェクトを再ビルドし、プロジェクトレス デバッグを再開し、CPU2 のターゲットを接続した後で正しい順序でプログラムをロードし、両方の CPU の IPCsync 関数に渡されるパラメータが同じであることも確認します。それ以外の場合は、CPU1 プロジェクト フォルダの sysconfig ファイルで CPU2 のブート モードが「フラッシュ バンク 3 からブート」に設定されていることを確認して、ファイルを保存します。

6 主な注意点

1. **ブート シーケンスの所有権:**

   CPU1 は常にマスタ コアであり、最初にブートする必要があります。CPU1 は CPU2 のリセットを制御し、システム全体の初期化 (PLL、クロック、ウォッチドッグ) を担当しています。コールド状態の CPU2 には CCS を直接接続することはできません。CPU1 が動作しており、CPU2 をリセット状態から解放している必要があります。
2. **CCS でのロード順序の重要性:**

   最初に CPU1 に接続してロードし、システムを初期化して CPU2 をブートできるようになるまで動作させ (または最低でも CPU2 をリセット状態から解除し)、次に CPU2 に接続してロードします。CPU1 がシステム クロックを初期化する前に CPU2 をロードすると、動作が不安定になります。
3. **CCS プロジェクトの分離:**

   デュアルコア アプリケーションは、2 つの完全に独立したイメージ (出力ファイル) です。これらは共有メモリと IPC を介してのみリンクされます。単一の「デュアルコア プロジェクト」は存在しません。(セクション 5 のビルドの後処理手順を使用する場合は除く)
4. **メモリの所有権 / アクセス権:**

   共有 SRAM ブロックには所有権の割り当てがあります。CPU1 が GS RAM ブロックを所有している場合、CPU2 がそのブロックに書き込みを試みるとメモリ アクセス違反が発生します。CPU1 と CPU2 のリンカ コマンド ファイル (.cmd) は、重複しないように慎重に調整する必要があります。また、共有セクションでは、専用の IPC/MSG RAM 領域、または適切に所有された GS RAM を使用する必要があります。
5. **一方のコア停止時の他方コアへの影響:**

   ブレークポイントで CPU1 を停止しても、CPU2 は、明示的に停止しない限り (または CCS で同期グループを使用しない限り)、動作を継続します。応答すべきコアが停止している場合、実行中のコア上の IPC フラグ ポーリング ループが無制限にスピンし続ける可能性があります。これは、デュアルコア デバッグ中に混乱を招く動作が生じる最も一般的な原因です。
6. **グローバル ブレークポイント / 同期グループ:**

   CCS はコアのグループ化をサポートしているため、一方のブレークポイントで両方が同時に停止します。IPC を多用するコードの場合、これはほぼ常に望ましい動作です。そうでないと、実行中のコアが先回りして IPC メッセージを消費してしまい、それを確認する頃には状態が不整合になってしまいます。
7. **ウォッチドッグ:**

   各コアには各自のウォッチドッグがあります。デバッグのために CPU1 を停止したが、CPU2 が実行中で共有ペリフェラルの処理中、またはその逆の場合、監視されていないコアのウォッチドッグがタイムアウトしてデバイスがリセットされることがあります。デバッグ ビルドの初期段階でウォッチドッグを無効化します。
8. **フラッシュと RAM の比較:**

   フラッシュ ビルドでは、両方のコアがフラッシュ コントローラを共有しており、同じパイプライン ステージで両方のコアから同時にフラッシュ読み出しが行われると、ストールが発生する可能性があります。デバッグ中は、これを回避し、消去サイクルなしで高速な再ロードを可能にするために、ほとんどのチームは RAM 上で作業を行います。フラッシュ プログラミングでは、両方のコアを調整する必要があります (通常、CPU1 がすべてのフラッシュの初期化を行います)。
9. **IPC 競合状態:**

   IPC フラグ ベースのプロトコルは、その性質上非同期です。よくあるバグ:CPU1 がフラグをセットしてすぐに共有バッファにデータを書き込むが、CPU1 の書き込み完了前に CPU2 がバッファ読み取りをしてしまう。IPC フラグは、データ書き込みが完了した後の通知にのみ使用し、必要に応じてメモリ バリア (C28x では __asm(" RPT #7 || NOP")) を使用します。
10. **CLA は 3 番目のデバッグ可能コア:**

    CLA を搭載したデバイスでは、CLA は CCS 上で個別のデバッグ ターゲットとして表示されます。CLA シンボルは CPU2.out からロードします (それらは一緒にコンパイルされます)。CLA には RTOS およびスタックがなく、デバッグの可視性が非常に制限されているため、ステップ操作やレジスタの確認は可能ですが、コール スタックはありません。
11. **異種混在デバッグ (F2838x CM コア):**

ARM CM コアは、まったく異なるツール チェーンを使用しています。CCS では、個別の ARM Cortex-M4 ターゲットとして表示されます。C28x プロジェクトと一緒に ARM コンパイラ プロジェクトが必要です。CM コアは、C28x ローカル RAM に直接アクセスすることはできません。すべてのデータ交換は MSG RAM を経由します。

7 デュアル コア書き込み用ワンクリック ソリューション

このセクションでは、2 つのバイナリ (CPU1 用 1 つ、CPU2 用 1 つ) を結合し、フラッシュ内の目的の領域に 1 手順で直接ロードする方法を説明します。各バイナリをそれぞれのコアに手動でロードするのではなく、結合されたバイナリを CPU1 にロードするだけにします。これは、ビルドの後処理手順を使用して実現します。

F28P65x で、一方の CPU にロードし、もう一方の CPU を直接制御するための結合出力ファイル生成する方法 (最新の C2000ware バージョンをダウンロードしていないユーザー向け)

- C2000 f28p65x から led_ex2_blinky_sysconfig_multi をインポートします。この操作で、CPU1 と CPU2 の両方のプロジェクトが自動的にワークスペースにインポートされます。
- マルチ フォルダを右クリックし、目的のビルド構成を選択します。
- CPU1 プロジェクト フォルダで **.sysconfig** ファイルを開き、CPU2 のブート モードを「Boot from Flash Bank 3」(Flash Bank 3 からブート) に変更して、ファイルを保存します。
- **sysconfig_multi** プロジェクトを右クリックし、「Build Project」(プロジェクトのビルド) を選択します。
- 次に、CPU2 を右クリック →「Properties」(プロパティ) →「Build」(ビルド) →「Steps」(ステップ) →「post-build step」(ビルドの後処理手順) の順に選択し、以下のコマンドを「postBuildStep」として手動で入力して保存します。コマンドは次のとおりです。

  ${CG_TOOL_ROOT}/bin/hex2000.exe --memwidth=16 --romwidth=16 -ii --map="..\..\combined.map" "..\..\led_ex2_blinky_sysconfig_cpu1\"FLASH_LAUNCHXL or FLASH"\led_ex2_blinky_sysconfig_cpu1.out" "..\..\led_ex2_blinky_sysconfig_cpu2\"FLASH_LAUNCHXL or FLASH""\led_ex2_blinky_sysconfig_cpu2.out" -o "..\..\combined.hex"
- CPU1 **targetConfigs** フォルダにあるターゲット構成ファイルを開き、「Start Project-less Debug」(プロジェクトレス デバッグを開始) をクリックします。
- ここで、CPU1 と CPU2 を右クリックして、ターゲットに接続します。
- CPU1 を右クリックし、「Flash setting」(フラッシュ設定) →「Flash Bank Map Settings」(フラッシュ バンク マップ設定) まで下にスクロールして、すべてが 0 に設定されていることを確認します (CPU1 自体からデータ全体をロードするので、すべてのバンクにアクセスできる必要があります)
- さらに下にスクロールして、「Erase settings」(設定消去) の下で「Entire flash」(フラッシュ全体) を選択します。
- 上にスクロールして「Configure Clock」(クロックの設定) をクリックしてから、保存して閉じます。
- 次に、「Run」(実行) →「Load program」(プログラムのロード) →「Browse」(参照) →「combined.hex」の順にクリックします
- 「View」(ビュー) →「Memory Browser」(メモリ ブラウザ) を開き、データが CPU1 では 0x8000 に、CPU2 では 0xE0000 に正しくロードされていることを確認します。
- CPU1 と CPU2 の両方で「Run」(実行) をクリックして実行を開始し、Launchpad の LED または出力コンソールを使用して出力を確認します。

最新の C2000ware バージョンをダウンロードしている場合 (次回のリリースで入手可能になる予定です)

マルチ プロジェクトを直接ビルドすると、マルチ プロジェクト フォルダ内に combined.hex ファイルと combined.map ファイルが作成されます。サンプルを実行するためには、上記のプロジェクトレス デバッグ手順に従ってください。

projectspec での postBuildStep のマージ:

次のファイルで以下に示す変更を行いました。

「examples_driverlib\c28x_dual\led\sysconfig_led_ex_f28p65x\CCS\led_ex2_blinky_sysconfig.projectspec」

これは、c2000-device-support リポジトリにあり、SDK にマージするために行われました。

Launchpad での検証では、次のファイルにも同じ変更を行う必要があります。

「C:\ti\c2000\C2000Ware_26_01_00_00\driverlib\f28p65x\examples\c28x_dual\led\CCS\led_ex2_blinky_sysconfig.projectspec」ファイル、これは c2000ware フォルダにあります。

- <configuration name="FLASH" compilerBuildOptions="--opt_level=off -I${C2000WARE_ROOT} -I${PROJECT_ROOT}/device -I${C2000WARE_DLIB_ROOT} --define=_FLASH -v28 -ml -mt --cla_support=cla2 --float_support=fpu64 --tmu_support=tmu1 --define=DEBUG --define=CPU2 --gen_func_subsections=on --

diag_warning=225 --diag_suppress=10063" linkerBuildOptions="--entry_point code_start --stack_size=0x3F8 --heap_size=0x200 --define=_FLASH "

postBuildStep="${CG_TOOL_ROOT}/bin/hex2000.exe --memwidth=16 --romwidth=16 -ii --map='..\..\led_ex2_blinky_sysconfig_multi\combined.map' '..\..\led_ex2_blinky_sysconfig_cpu1\FLASH\led_ex2_blinky_sysconfig_cpu1.out' '..\..\led_ex2_blinky_sysconfig_cpu2\FLASH\led_ex2_blinky_sysconfig_cpu2.out' -o '..\..\led_ex2_blinky_sysconfig_multi\combined.hex'"

/>

<configuration name="FLASH_LAUNCHXL" compilerBuildOptions="--opt_level=off -I${C2000WARE_ROOT} -I${PROJECT_ROOT}/device -I${C2000WARE_DLIB_ROOT} --define=_FLASH -v28 -ml -mt --cla_support=cla2 --float_support=fpu64 --tmu_support=tmu1 --define=DEBUG --define=CPU2 --gen_func_subsections=on --diag_warning=225 --diag_suppress=10063" linkerBuildOptions="--entry_point code_start --stack_size=0x3F8 --heap_size=0x200 --define=_FLASH "

postBuildStep="${CG_TOOL_ROOT}/bin/hex2000.exe --memwidth=16 --romwidth=16 -ii --map='..\..\led_ex2_blinky_sysconfig_multi\combined.map' '..\..\led_ex2_blinky_sysconfig_cpu1\FLASH_LAUNCHXL\led_ex2_blinky_sysconfig_cpu1.out' '..\..\led_ex2_blinky_sysconfig_cpu2\FLASH_LAUNCHXL\led_ex2_blinky_sysconfig_cpu2.out' -o '..\..\led_ex2_blinky_sysconfig_multi\combined.hex'"

/>

上記の変更は、CPU2 プロジェクトのビルドの後処理手順として行われました。

f28p65 で単一出力ファイルがどのようにロードされるか

- hex2000 は、2 つの CPU 出力ファイルを結合し、最初のファイルから EOF レコードを自動的に削除するため、2 番目のファイルは途切れることなく、単一の Intel-HEX イメージとして続きます。
- 結果として得られる HEX は、0x08000 (Flash Bank 0) に CPU 1 のコードおよび 0xE0000 (Flash Bank 3) に CPU 2 のコードを含んでおり、1 回のフラッシュ操作で両方のバンクが一緒にプログラムされます。
- ロード後、開始ベクトルが再設定されます。CPU 1 は自身のフラッシュ アドレスから実行を開始し、CPU 2 のブートを解除します。CPU 2 は、Flash Bank 3 からブートします。
- CPU 1 には 5 つのフラッシュ バンクすべてに対する書き込みアクセス権が付与され、自身の領域と CPU 2 で使用される領域をプログラムするための完全な制御が可能になります。
- ロードは、単一の **Run → Load Program** コマンドで行われます。ロード後、「Memory Browser」(メモリ ブラウザ) を開いて、CPU 1 の場合は 0x08000、CPU 2 の場合は 0xE0000 のデータを確認できます。
- 実行が開始されると、CPU 1 が停止し、CPU 2 のブートが開始され、CPU 2 が動作を開始してからのみ処理が進められます。これにより、協調したデュアル CPU 動作が保証されます。

この結合 hex アプローチが他のデュアル コア デバイスでは機能しないのはなぜか

CPU1 と CPU2 バイナリを結合する同じアプローチは、メモリの所有権が異なる場合、機能しないことがあります。たとえば、F2838x デバイスの場合を考えます

f2838x フラッシュ メモリ:

CPU	物理 / グローバル フラッシュ アドレス	サイズ	ローカル アドレス
CPU1	0x200000	512KB	0x080000 (CPU2 も参照している同じアドレス)
CPU2	0x300000	512KB	0x080000 (CPU1 も参照している同じアドレス)

- ここでは、両 CPU がそれぞれのローカル メモリ マップを持っており、各 CPU は各自のフラッシュ開始位置を **0x080000** として認識しています。それが、単一 CPU ビルドから生成されたリンカ マップで、両方の CPU が同じアドレスから開始しているように見える理由です。

- **hex2000** は、.out ファイルに表示されるローカル アドレスに基づいてファイルを結合します。両方の CPU が 0x08000 をフラッシュの開始と報告するため、アドレスの再マッピングを行わずに結合すると、一方のイメージが他方のイメージに上書きされます。
- 正しくプログラムするには、結合する前に CPU2 セクションをグローバル ベースに再マッピングする必要があります。

現在、他のデュアル コア デバイスに対して同様のソリューションを実装する作業を進めています。

8 プロセッサ間通信

IPC は、コアが互いに信号伝達し、データを渡すハードウェア メカニズムです。IPC の主要なコンポーネントは次のとおりです。-

- **IPC フラグ レジスタ**:コアのペアごとに 32 のハードウェア フラグ ビット (CPU1↔CPU2、CPU1↔CM、CPU2↔CM)。一方のコアがフラグをセットし、もう一方のコアが割り込みを取得、またはフラグをポーリングできます。これは軽量のシグナリング (「データが準備完了」、「コマンドを受信」など) に使用されます。
- **メッセージ RAM (MSG RAM)**:これは両方のコアから物理的にアクセスできる専用の SRAM であり、データを渡すために使用されます。各方向には個別の MSG RAM があります (CPU1 から CPU2 への MSG RAM は、破損を防止するため CPU2 の側からの書き込みが保護されています)。
- **グローバル共有 RAM (GS RAM)**:大容量の共有メモリ ブロックで、複数のコアからアクセスできます。このメモリの所有権は動的に割り当てることができます。C28x コアと CM コアの間では利用できません (そこでは MSG RAM のみが使用されます)。
- **IPC ドライバ / driverlib**:C2000Ware は、ハードウェア フラグの上でメッセージ送受信プロトコル (コマンド + アドレス + データ) の層を構成するソフトウェア IPC ドライバを提供しています。

各デバイスの特定の IPC レジスタの詳細については、デバイスの TRM を参照してください。

9 役立つリンク

C28x デバイスの一般的なデバッグのヒント —『C2000 デバッグのヒントとコツ』

マルチ コアのデバッグ —『6. マルチ コアのデバッグ — C2000™ マルチコア開発ガイド』

関連する FAQ (よくある質問)

『(4) TMS320F28388D：2 コア付きデバッガを使用 — C2000 マイコン フォーラム — C2000™ マイコン — TI E2E サポート フォーラム』

『(4) TMS320F28384S：マルチ コアのデバッグに関する問題 — C2000 マイコン フォーラム — C2000™ マイコン — TI E2E サポート フォーラム』

『(4) TMS320C28345：C28p65 デュアル コア CPU2 のデバッグに関する問題 — C2000 マイコン フォーラム — C2000™ マイコン — TI E2E サポート フォーラム』

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。

最終更新日：2025 年 10 月