ANALOG DEVICES

AN-1267
アプリケーション・ノート

ADSP-CM408F ADC コントローラを使用した モーター・コントロール帰還サンプリング・タイミング

著者: Dara O'Sullivan、Jens Sorensen、Aengus Murray

はじめに

このアプリケーション・ノートでは、高性能モーター・コントロール・アプリケーションでの電流帰還システムの妥当性と有効性を中心に、ADSP-CM408F A/D コンバータ・コントローラ（ADCC）ブロックの主な機能を紹介します。

このアプリケーション・ノートの目的は、A/D コンバータ（ADC）モジュールの主要機能を説明すること、およびモーター・コントロール・アプリケーション用の設定についてガイダンスを提供することです。アナログ・デバイセズが提供する ADCC ドライバの使い方を説明するコード・サンプルも記載してあります。

この ADCC のすべての機能、コンフィギュレーション・レジスタ、アプリケーション・プログラム・インターフェース（API）の詳細については、ADSP-CM402F/ADSP-CM403F/ADSP-CM407F/ADSP-CM408F 製品ページおよび ADSP-CM40x Mixed-Signal Control Processor with ARM Cortex-M4 and 16-bit ADCs Development Products 製品ページで提供している ADSP-CM40x Mixed-Signal Control Processor with ARM Cortex-M4 Hardware Reference Manual を参照してください。

このアプリケーション・ノートでは電流帰還を中心に述べていますが、これと同様の設定とアプリケーションの原理は、他の信号の帰還と測定にも適用することができます。

同様に、このアプリケーション・ノートは ADSP-CM408F に焦点を当てていますが、その原理は通常、ADSP-CM402F/ADSP-CM403F/ADSP-CM407F/ADSP-CM408F ファミリー内の他の製品にも適用することができます。

Rev. A

アナログ・デバイセズ株式会社

本　　社／〒105-6891　東京都港区海岸 1-16-1　ニューピア竹芝サウスタワービル
電話 03（5402）8200
大阪営業所／〒532-0003　大阪府大阪市淀川区宮原 3-5-36　新大阪トラストタワー
電話 06（6350）6868

目次

改訂履歴

9/14—Rev. 0 to Rev. A

Changes to Introduction Section....1
Changes to Figure 2....3
Changes to Figure 3....4
Changes to Current Feedback Scaling Section....5
Changes to ADC Operational Timing Section....7
Added Adjustment of Sampling Instant Section and Figure 11; Renumbered Sequentially....8
Added Trigger Routing for Enhanced Precision Sample Timing Section....12
Changes to Example Code Section....13

9/13—Revision 0: Initial Version

電流帰還システムの概要

図 1 に、モーター・コントロール・アプリケーションでの電流帰還の例を示します。この構成は、インバータのローサイドのフェーズ・レグではなく、モーターの相巻線電流をサンプリングする高性能モーター駆動では一般的です。中電流レベルから高電流レベルでは、電流測定パスに電流トランスデューサまたは電流トランス（CT0 および CT1）を使う必要があります。これは抵抗電流シャントが大きくなり効率が悪くなるためです。

図 1 のセットアップでは、プロセッサは安全な絶縁バリアの低電圧側に存在し、信号アイソレーションは CT0 と CT1 にもともと備わっており、デジタル・アイソレーションもマイクロプロセッサのパルス幅変調（PWM）出力とゲート・ドライバの間に存在しています。

一般に、電流トランスデューサ出力と ADC 入力との間でシグナル・コンディショニングを行う必要があります。その目的は、信号範囲を一致させること、ならびに高周波ノイズをフィルタリングすることにあります。コンディショニングされた電流測定信号は ADC に入力され、サンプリングと変換が行われます。1 つの巻線電流測定値を各 ADC 入力に供給すると、複数の電流測定値の同時サンプリングが可能になるため、制御ループの精度が向上し、性能が強化されます。さらに、このサンプリングと PWM 同期パルスとの同期も、ハードウェアで直接設定することができます。

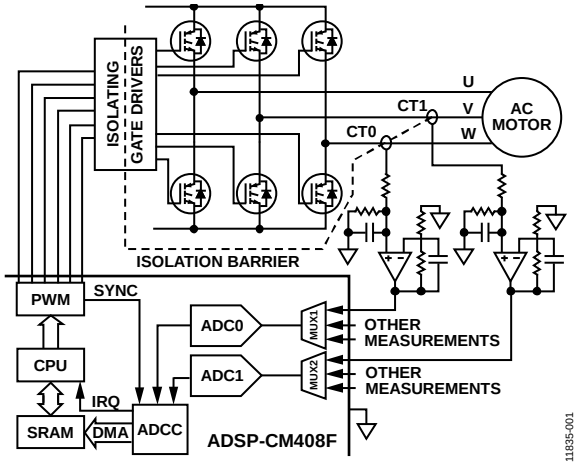

図 1.モーター・コントロールでの
ADSP-CM408F ADC への電流帰還

これらの機能を使うと、相電流を測定する PWM サイクル内のタイミング・ポイントを正確に決めることができます。この測定タイミングをゼロ・ベクタの中心値または PWM サイクルの中心値に合わせると、電流をサンプリングするレベルが実質的に瞬時平均電流と一致するようにできるため、スイッチング・リップルが無視されます。

図 2 に、ゼロ・ベクタ中心値と PWM サイクル中心値で発生する W 相と V 相の同時サンプリングを示します。

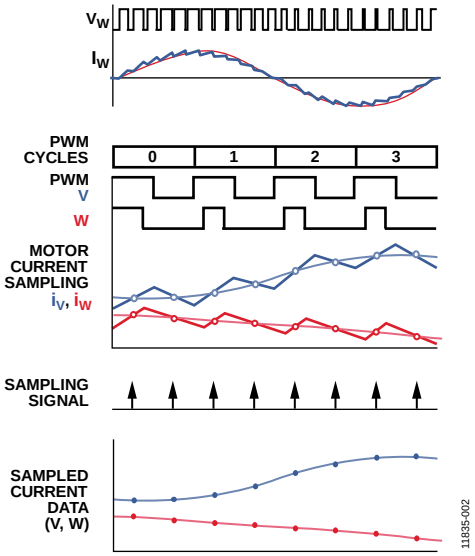

図 2.平均電流サンプリングの説明

データの変換が完了すると、そのデータをダイレクト・メモリ・アクセス（DMA）を使ってコントローラのスタティック・ランダム・アクセス・メモリ（SRAM）へ転送することができ、転送が完了すると割込みが発生します。コア・モードでメモリ・マップド・レジスタを使って ADC のステータスとデータの直接読出しも可能ですが、この方法ではプロセッサの負荷が増えます。

一般に、DC バス電圧、絶縁型ゲート・バイポーラ・トランジスタ（IGBT）の温度、モーター位置の正弦出力および余弦出力のような、その他のアナログ信号もサンプリングされます。このアプリケーション・ノートでは電流帰還に焦点を当てていますが、大部分の情報はシステム内の関連するその他の測定パラメータにも適用することができます。

ADCモジュールの概要

この ADC は、最大 14 ビット精度の 2 個の 16 ビット高速低消費電力逐次比較レジスタ（SAR）を内蔵した設計になっています。

入力段にはマルチプレクサがあるので、独立して制御される 2 個の ADC に対して最大 26 本のアナログ入力源を組み合わせることが可能です（内訳は、ADC 1 個につき、アナログ入力が 12 本と DAC ループバック入力が 1 本）。任意のタイミングで 2 チャンネルが同時にサンプリングされます。ADC 変換時間は、最小 380 ns です。シングルエンド・アナログ入力の電圧入力範囲は、0 V ～ 2.5 V です。

マルチプレクサと ADC の間にバッファが内蔵されているため、ADSP-CM408F の外部にシグナル・コンディショニングを追加する必要性が低減されます。さらに、各 ADC に 2.5 V のリファレンスを内蔵しています。外付けリファレンス電圧が必要な場合、この内蔵リファレンスを上書き駆動することができます（ADCC_CFG レジスタを使ってこのオプションを選択）。

ADSP-CM408F 内部のアナログ・サブシステム全体の概略図を図 3 に示します。ADSP-CM408F は、複数のチップを混載したシステム・イン・パッケージ（SiP）であり、ADC シリコンはプロセッサ・シリコンと異なるプロセスで製造されています（図 3 参照）。

ADCC は、ADC 内部のタイミングとプロセッサとの同期を行い、サンプリングされたデータを SRAM へ DMA 転送する際の管理を行います。

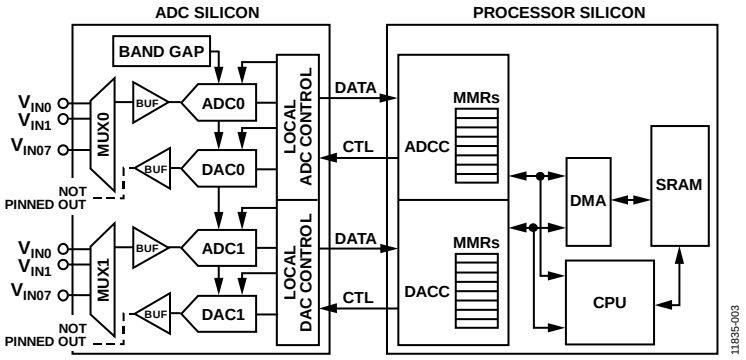

図 3.ADSP-CM408F アナログ・サブシステム

電流帰還のスケーリング

最大入力範囲で ADC 機能を正しく使うためには、帰還信号を正しい方法でスケールすることが重要です。帰還パスでの信号の進行を図 5 に示します。相巻線のバイポーラ電流 I_W は、電流トランスデューサ（または電流トランス）とシグナル・コンディショニング回路の機能的組み合わせにより、ADC 入力でのユニポーラ電圧に変換されます。

電流トランスデューサの伝達関数は、次式で表されます。

$$V_{IW} = K_{CT}I_W + V_{0CT}$$

ここで、
V_{IW} は出力電圧。
K_{CT} はトランスデューサのリニア・ゲイン係数。
V_{0CT} はトランスデューサのゼロ電流オフセット電圧。

K_{CT} は、トランスデューサの種類によっては何らかの電流レベルで非直線性を持つ傾向があるので、精度を上げるためには、I_W の関数として、すなわち K_{CT}（I_W）として表す必要があります。ADC 入力電圧は次のように表されます。

$$V_{IW\_ADC} = K_{SIG}V_{IW} = K_{SIG}[K_{CT}\ (I_W)\ I_W + V_{0CT}]$$

ここで、
K_{SIG} はシグナル・コンディショニング回路の低周波ゲイン。

このユニポーラ電圧は 16 ビット符号なし整数に変換されます。これがプロセッサ・メモリへ DMA 転送され、その後、新しいデータ・サンプルが使用可能になったことを制御プログラムへ通知する割込みが発生します。ADC の理想的な伝達関数は次式で表されます。

$$N_{IW} = K_{ADC}V_{IW\_ADC} = \frac{2^{16}}{2.5}V_{IW\_ADC}$$

ここで、
N_{IW} は ADC のデジタル出力ワード。
K_{ADC} は、ADC のリニア・ゲインであり、式に示したように ADC 分解能を入力電圧範囲で除算した値です。

ADC 出力にはオフセットが発生します。ソフトウェア内に、オフセット補償値 N_{ADC_OFFSET} を含めることは一般によく行われています。ADC 自体のオフセット、およびトランスデューサとシグナル・コンディショニングから生ずる残留オフセットを補償するため、ADC 出力からこの補償値が減算されます。この値は、システム・スタートアップ時または駆動出力のディスエーブル時のようなゼロ電流区間にダイナミックに更新することができます。

最後に、電流トランスデューサのゼロ電流オフセット電圧 N_{CT_OFFSET} のデジタル値が ADC 出力から減算されて、符号付き値 IW が得られます。この IW は、実際の相巻線電流と次式の関係にあります。

$$IW = K_{ADC}\ (K_{SIG}[K_{CT}\ (I_W)\ I_W + V_{0CT}])\ - N_{ADC\_OFFSET} - N_{CT\_OFFSET}$$

ここで、

$$N_{CT\_OFFSET} = \frac{2^{16}}{2.5}V_{0CT}$$

この符号付き 16 ビット値は、コントローラの構成に応じて、浮動小数点値に変換することも、また直接使用することも可能です。最適なフル ADC 範囲を実現するためには、システム内の正のピーク制御電流が 2.5 V の ADC 入力電圧に対応していること、および負のピーク制御電流が 0 V の ADC 入力に対応していることが必要です。

この例を図 4 に示します。この図では、代表的な電流波形と、それに対応するゼロ・レベル、ピーク・レベル、公称レベルを示してあります。図 4 の電流レベルはスケーリングされた数値に変換され（表 1 参照）、これが信号測定システムまで送られます。これを図 5 に示します。

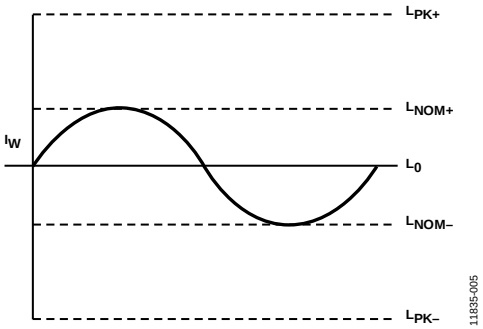

図 4. 電流帰還信号の振幅

表 1. 電流帰還信号の振幅

Level	I_W (A)	V_{IW} (V)	V_{IW_ADC} (V)	N_{IW}
L_{PK+}	6.8	4.625	2.313	0xECD9
L_{NOM+}	4	3.75	1.875	0xC000
L_0	0	2.5	1.25	0x8000
$L_{NOM–}$	−4	+1.25	+0.625	0x4000
$L_{PK–}$	−6.8	+0.375	+0.188	0x1340

この例では、LEM®社の CAS 6-NP ホール効果電流トランスデューサ（1 次側の巻数は 3 ターン、出力電圧は 0 V ～ 5 V）の後にゲイン = 0.5 のシグナル・コンディショニング回路を接続しています。

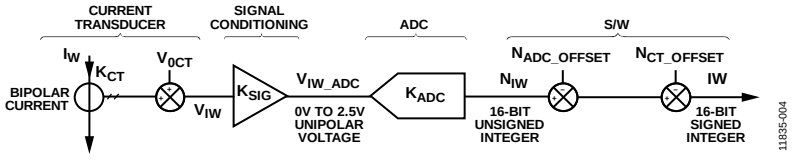

図 5. 電流帰還パスでの各スケーリング

ADCタイミングの考慮事項

サンプリング・イベントと PWM サイクルとの同期は、正確な電流帰還にとって重要です。PWM サイクルを基準とする ADCC の動作シーケンスの概念図を図 6 に示します。次の一連のイベントは PWM 同期パルスでトリガされます。

1. PWM 同期パルスがタイマーの開始をトリガします。
2. ADCC はイベント情報からのサンプリング時間とタイマー時間を連続的に比較します。
3. タイマーの一致が生じ、ADCC が ADC の動作を設定します。
4. ADC が使用可能な場合、ADCC はイベント情報を使って適切なチャンネルを選択します。
5. ADCC は ADC 変換シーケンスをトリガし、ADC はデータをサンプリングして変換します。
6. データが ADCC へ戻されます。
7. ADCC は DMA （LSB ファースト）を使ってデータをメモリ・ロケーションへ転送します。
8. データ・サンプルが使用可能であることを CPU に通知する割込み（IRQ）が発生します。

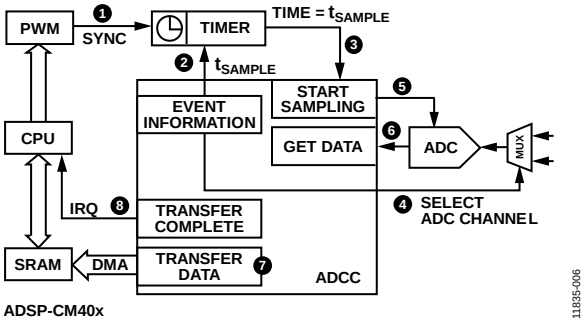

図 6. ADCC の動作シーケンス

ADCC イベントのタイミング

コントローラは、最大 24 個のサンプリング・イベントの設定とタイミングを管理します。これらのイベントのタイミングは、2 個のタイマーの内の 1 つ（TMR0 または TMR1）を開始するトリガと、タイマー開始後のイベント時間による制約を受けます。

図 7 に示すように、トリガ・ソースは、PWM 同期パルス、タイマー、または I/O ピン割込みなど、様々なペリフェラル・イベントまたはプロセッサ・イベントから選択できます。各イベントは、イベント番号 Event x、イベント時間 TIMEx、制御情報 CTLx、結果データに対応します。図 7 で CTLx と表示したイベント制御情報には、ADC インターフェース、ADC チャンネル番号、使用される ADC タイマー、同時サンプリングの選択、イベントに対応した ADC データのメモリ・オフセットのような、各サンプル・イベントの情報が含まれます。ADCC はこの情報を使って、正しい ADC チャンネル CHx をマルチプレクスし、ADC 変換を開始し（CVST0/CVST1 信号）、正しいデータを該当するイベント・データ・レジスタへ転送します。

次に、DMA 転送を設定して、各イベントの ADC データを SRAM へ転送します。すべてのイベントと後続の DMA 転送が完了すると、新しい ADC データが使用可能であることをメイン・アプリケーション・コードへ通知する割込みが発生します。

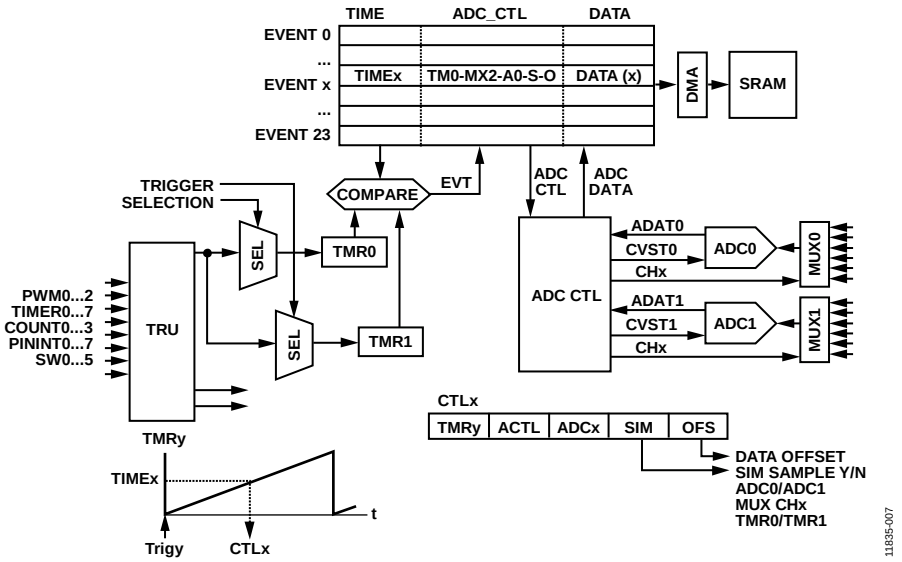

図 7. ADCC モジュールの機能図

例として、図 8 に ADC タイマー0 に対応した 3 つのサンプリング・イベントを示します。PWM 同期パルスがタイマーのトリガであり、各イベントにはイベント時間が関連付けられています。イベント 0 とイベント 1 は、同時サンプリング・イベントであり、イベント時間レジスタのイベント時間はゼロに設定されています。イベント 0 およびイベント 1 の後、イベント 2 も同じように、イベント 2 の時間レジスタで指定される時間（ADC クロック周期（t_{ACLK}）の倍数で表される時間）に発生します。イベント 2 がタイマー0 に対応する最終イベントである場合は、電力を節約するため、イベントの処理後にタイマーは動作を停止します。

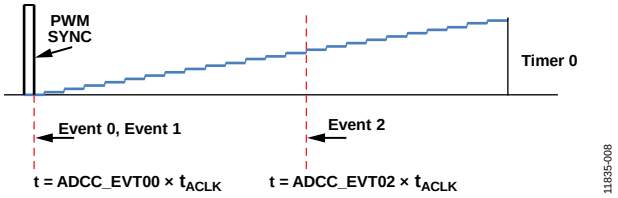

図 8.イベントのタイミング

ADC 動作タイミング

ADCC コントローラからサンプリング・イベントがトリガされた後、ADC の動作自体に関係して変換時間遅延が生じます。これを図 9 に示します。状況としては、1 つの ADC イベントが各 ADC インターフェースに関連付けられており、かつ 2 つのイベントの同時サンプリングがイネーブルであるというものです。

ADC のこの動作には、次の 3 つの変換サイクルが関係します。

1. 読出し対象 ADC チャンネルを選択する 8 ビット制御ワードの書込み（ADCC_EVTCTL.CTLWD）
2. ADC のサンプリングと変換をイネーブルする変換パルスをアサート
3. 16 ビット ADC データの ADCC への読込み

ADCC は、これら 3 つのイベント・フェーズでチップ・セレクトとゲーティド・クロック信号を供給します。ADC と ADCC とのインターフェースは、デュアル・ビット・オプション付きのシリアル・インターフェースです。このため、各 CS パルス区間で供給される最小クロック・サイクル数（ADCC タイミング・コントロール・レジスタ・フィールド NCK）は 8 になります。その他の重要な設定は、ADC クロック周波数、変換サイクル・チップ・セレクト（t_{CSCS}）間の最小遅延（ACLK サイクル数）、CS エッジと ACLK エッジ（t_{CSCK} と t_{CKCS}）間の最小遅延です。したがって、1 対の同時サンプリングされる信号に対する ADC 変換サイクル時間 t_{CONV_ADC} は次式で与えられます。

$$t_{CONV\_ADC} = \frac{3}{f_{ACLK}}\left(t_{CSCK} + NCK + t_{CKCS} + t_{CSCS}\right)$$

ここで、f_{ACLK} は ADCC クロックの周波数。

ADCC クロックは、タイミング・コントロール・レジスタ ADCC_TCA の分周比 ACKDIV を使って内部でプロセッサ・システム・クロック（f_{SYSCLK}）から発生され、次式で計算されます。

$$f_{ACLK} = \frac{f_{SYSCLK}}{ACKDIV + 1}$$

次にシステム・クロックが、プロセッサ・コア・クロック（$f_{CORECLK}$）から発生されます。最適システム性能は、$f_{CORECLK}$ が f_{SYSCLK} の整数倍のとき実現されます。ADC 変換が完了すると、ADC データのデータ・メモリへの DMA 転送で遅延が加わり、最後に、メイン・アプリケーション・プログラムでデータ・フレームを使用できるようにする割込み要求の処理で遅延が加わります。このため、トリガ（例えば PWM 同期パルス）からアプリケーションでデータが使用可能になるまでの合計時間は次式で与えられます。

$$t_{CONV\_TOTAL} = t_{CONV\_ADC} + t_{DMA} + t_{IRQ}$$

ここで、
t_{DMA} は DMA 転送の平均時間、
t_{IRQ} は割込み要求サービスの平均時間。

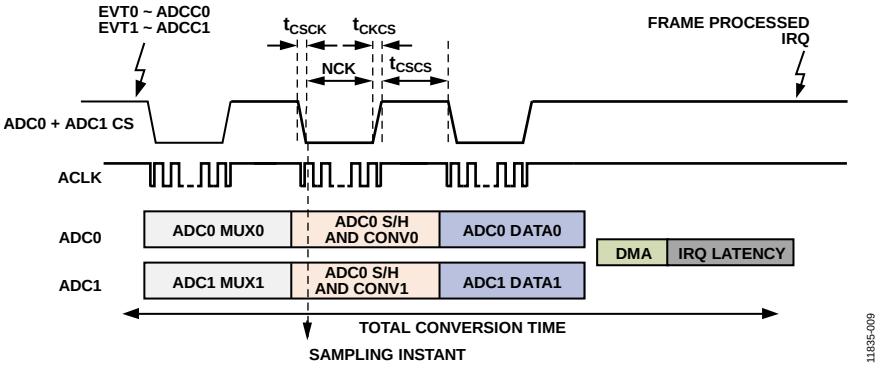

図 9. 1 イベント同時サンプリングの変換タイミング

代表的なタイミング設定を表 2 に示します。時間に対する幾つかの制約も示します。ADC の正しい性能を実現する絶対的制約事項は、ADC サンプリングと変換サイクル（$t_{CONV\_ADC}/3$）に少なくとも 380 ns を許容することです。1 イベント同時サンプリング結果のタイミングをモーター巻線電流のサンプリングを基準として図 10 に示します（この図は説明上、強調してあります）。

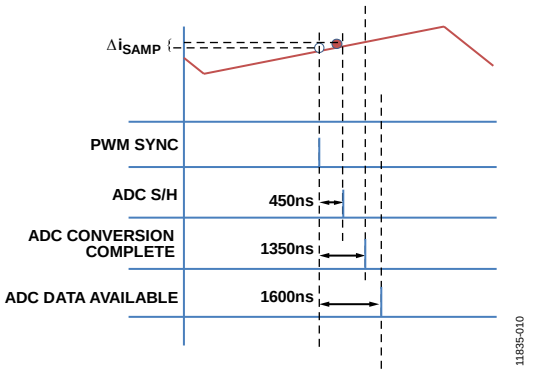

図 10.サンプル遅延時間

これらの設定で、電流波形上、所望するサンプリング・ポイントと実際のサンプリング・ポイントの間に 450 ns のオフセットがあります。これは、1 チップ・セレクト・パルス幅（200 ns + 25 ns + 0 ns）+ チップ・セレクト間の 1 パルス幅（225 ns）に一致します。これにより、平均モーター巻線電流と実際にサンプリングされる電流との間の差 Δi_{SAMP} が発生し、代表的な電流制御ループ帯域幅が 1 kHz、すなわち位相シフトが 0.2°以下としても、この差をサンプリング・タイミングのスケジューリングで考慮することが必要です。さらに、代表的な PWM 周波数 = 10 kHz の場合、ADC データは表 2 の設定値に対する PWM 同期パルスの発生から 2%以下の有効 PWM サイクル時間以内にアプリケーション・プログラムから使用可能になります。イベント発生時に ADC がアイドル状態にある場合、イベントがアクティブになってから ADC 動作が開始されるまでに 4 ～ 5 SYSCLK サイクルの追加遅延が発生します。

サンプリング・タイミングの調整

重要なのは、モーター電流サンプリングのタイミング精度をさらに上げることと、必要なサンプリング・タイミングと実際のサンプリング・タイミング間の 450 ns のオフセットを除去することです。こうした精度向上が特に有益なのは、低インダクタンス・サーボ・モーターのような使用事例や、比較的高いスイッチング周波数が使用されているような状況です。この小さい時間オフセットを相殺させる 1 つの選択肢は、汎用（GP）タイマーを使用して、PWM 同期パルスの 1 ADCC チップ・セレクト・パルス幅だけ前のタイミングにトリガを発生させることです。これは、図 11 に示すように、前の PWM 同期パルスから GP タイマーをトリガして実現することができます。

この方法では、PWM サイクルの終わりに向かってサンプリング・イベントをスケジューリングする際に注意が必要です。すべてのサンプリング・イベントは、次のサイクル（図 11 で EVT0 と表示）の開始より 1 チップ・セレクト・パルス幅だけ前に完了する必要があります。

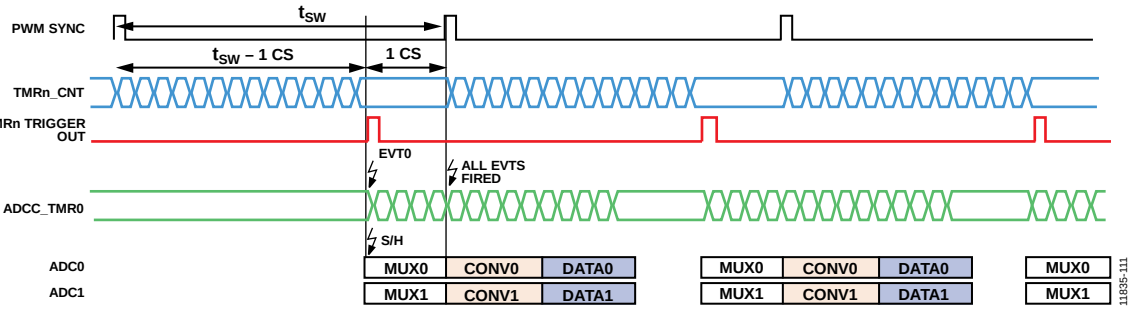

図 11. サンプリング・タイミングの調整

表 2. 代表的な ADC セットアップのタイミング設定値

Parameter	Value	Comment	Set By
$f_{CORECLK}$	240 MHz	Maximum allowed	PLL configuration
f_{SYSCLK}	80 MHz	Maximum is 100 MHz	$f_{SYSCLK} = f_{CORECLK}/3$
f_{ACLK}	40 MHz	Maximum specified is 50 MHz	ADCC_TCA0.CKDIV = 1
CS Time （t_{CSCS}）	200 ns	Must allow sufficient ACLK cycles for transfer of CTL word and data	ADCC_TCA0.NCK = 8
CS Edge to ACLK Edge （t_{CSCK}）	25 ns	Minimum time at 40 MHz, recommended	ADCC_TCB0.TCSCK = 1
ACLK Edge to CS Edge （t_{CKCS}）	0 ns	Recommended	ADCC_TCB0.TCKCS = 0
Time Between CS （t_{CSCS}）	225 ns	Must be >150 ns for accurate sampling	ADCC_TCB0.TCSCS = 9
t_{CONV_ADC}	450 ns		
t_{DMA}	50 ns	On average takes 4 SYSCLK cycles	
t_{IRQ}	200 ns	On average takes 16 SYSCLK cycles	

ADC パイプライン化

新しいイベントが ADC で処理中の既存イベントと重複して始まると、ADCC は新しいイベントを深さ 8 の FIFO バッファにペンディング・イベントとして格納します。このバッファの 1 つが各 ADC インターフェースから使用可能です。アクティブ・イベントに対して制御ワードが書き込まれると、ADCC は直ちに先頭のペンディング・イベントに対して制御ワードの書込みを開始し、同時にアクティブ・イベント・サンプリング・フェーズが始まります。同様に、先頭のペンディング・イベントに対する制御ワード・フェーズが完了すると、2 番目のペンディング・イベントが対応する制御ワード・フェーズを開始させます。この方法では、ADCC は 3 つのパラレル・イベントを各 ADC インターフェースにパイプライン方式でインターリーブすることができます。このため、イベントの間隔をコンパクトで効率よい方法で決めることができます。

このようにイベントがパイプライン化されるようイベント・タイミングを構成すれば、最高の ADC スループットが得られます。このパイプライン化を図 13 に示します。この図では 3 対の同時サンプリング・イベントが互いに近いタイミングでトリガされています。ADCC はイベント 0 とイベント 1 の処理を開始し、同時にイベント 2～イベント 5 を FIFO へ格納します。後で、ADC リソースが使用可能になったときこれらのイベントを処理します。

図 7 を見てわかることは、各イベントの様々なステージにある 6 つすべてのイベントが CS アサーションのいずれかの最中に ADCC によって処理されること、および連続するサンプル間の間隔がわずか 18 ACLK サイクルであることです。表 2 の設定値に対してこの時間間隔は 450 ns に対応し、ACLK 周波数を上げてさらに小さくすることができます。モーター・コントロール・アプリケーション内で ADC 帯域幅を最大化するための最良の方法は、サンプリング・イベントに関係するすべての PWM サイクルを慎重にパイプライン化することです。この方法では、新しい ADC サンプルが PWM サイクル内の最も早いタイミングで使用可能になります。図 13 に示すパイプライン化では、すべてのイベント時間をゼロに近づけること、すなわち PWM 同期パルスの直後にすることが必要とされます。

正しいスケジューリングを可能にするためには、イベント時間レジスタ ADCC_EVTnn （nn は 0～24 のレジスタ数）に格納されているイベント時間の間隔を最小 1 ACLK サイクルにすることが推奨されます。表 2 に示すタイミング設定値を持つ様々な同時サンプル対数に対して、開始遅延、DMA 転送、割込みサービスを含む、パイプライン動作での合計変換時間を図 12 に示します。

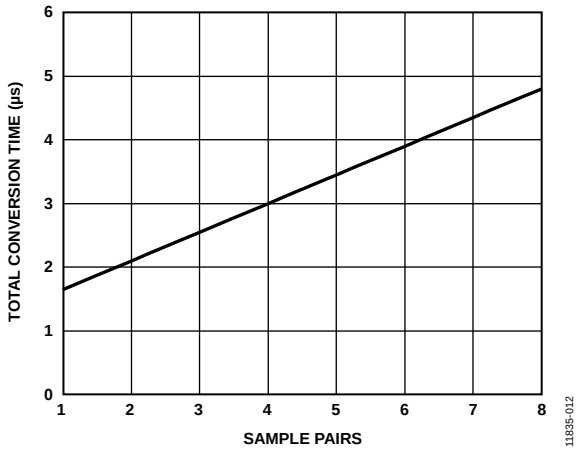

図 12.様々なサンプル対数に対する合計変換時間

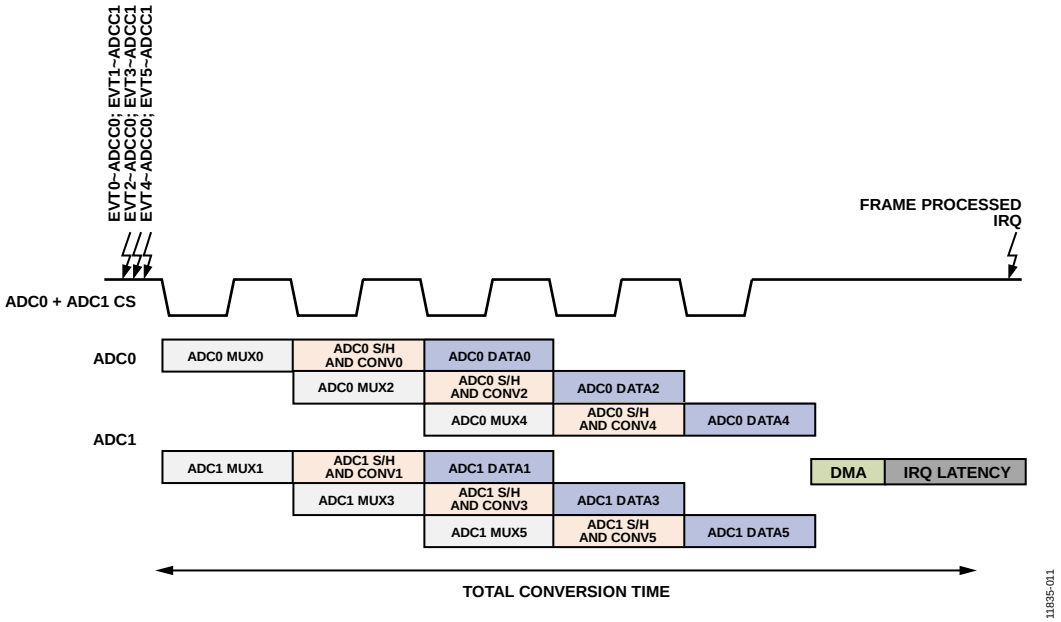

図 13.
ADC 内部のイベントのパイプライン化

ADCデータのアクセス

これまでに示してきた例はすべて、自動 DMA 転送を使ってメモリ内の ADC データにアクセスすることを前提としていました。ADCC メモリ・マップド・レジスタ（MMR）のコア読出しによる直接データ・アクセスも可能です（図 14 参照）。図 14 の ACK はアナログ・クロックではなく、アクノリッジ信号を表すことに注意してください。

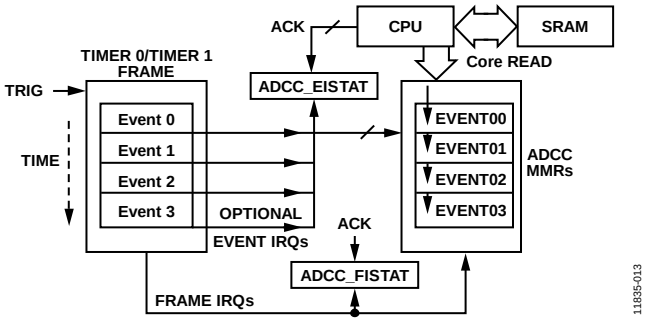

図 14.コア・モードでの ADC データ・アクセス

コア・モードでは、個別にマスクまたはアンマスクできるイベント割込みまたはフレーム割込みを使って新しいデータの準備ができたことを CPU に知らせます。このモードでの優れた柔軟性は、イベントのフレーム全体が完了する前に、個別イベントが完了した後直ちに読出し可能なことです。コア・モードの欠点は、割込みサービスと MMR 読出しアクセスを含む全体遅延が DMA モードより大きくなることです。コアとクロック比の最適な設定値では、各割込みサービスに対応する遅延に加えて、各 MMR の読出しに 10～12 SYSCLK サイクルを要します。

DMA モードでのデータ・アクセスを図 15 に示します。このケースでは、DMA 転送はタイマー・フレームの完了後にのみ実行され、フレーム割込みはこの DMA 転送が完了した後に CPU に通知されます。

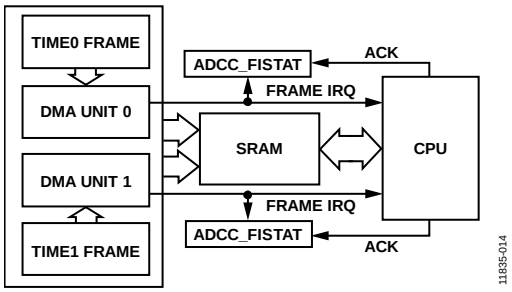

図 15.
DMA モードでの ADC データ・アクセス

両ケースとも、イベント割込みとフレーム割込みがアクティブである限り、その割込みのステータスは EISTAT レジスタと FISTAT レジスタから読み取れますが、次のトリガが発生する前に必ず、対応するビットをクリアするという方法で CPU がその割込みをアクノリッジする必要があります。そうしないとトリガ・オーバーラン状態がフラグ表示されます。

ADCC データ・フォルト検出

ADCC は、ADCC イベント・タイミングの不正なセットアップ、および／または非決定性イベント・シーケンスから発生するデータ・フォルトの発生時にセットされる多数のエラー・ステータス・レジスタ・ビットを持っています。これらのフォルトにより ADCC が過負荷になるか、無効な ADC データが発生します。次のようなデータ・フォルトがあります。

- トリガ・オーバーラン。現在のフレームが完了する前に次のトリガが発生。
- DMA 帯域幅。ユーザー定義の時間よりフレームの完了が長くなる。
- メモリ・エラー。ADC データ書込みに失敗。
- イベントの衝突。既存イベントの処理中に新しいイベントが発生。
- イベント・ミス。イベントが処理されなかった。

これらのすべてのエラーは、必要に応じてコアへの割込みソースとして設定可能です。これらのすべてが ADCC_ERRSTAT レジスタのビットをセットします。モーター・コントロールでは、特に電流帰還測定では、イベント・ミス、メモリ、トリガ・オーバーランに関係するエラーは、コア・アプリケーションでのモニタに重要です。これは、電流ループ・データの不正または喪失により制御ループが不安定になるためです。イベントの衝突はパイプライン動作では普通に発生するため、FIFO がフルにならない限り深刻な問題にはなりません。

ADCCモジュール、トリガのルーティング、メモリのセットアップ

ADC が使用可能になるまでに、ADCC モジュール、トリガ・ルーティング・ユニット、データ・バッファの設定に多数のステップがあります。設定が終わったら、DMA データ・アクセス・モードを使用するものとすると、DMA エンジンは自動的にプライマリ ADC データをメモリへ転送します。このデータはメイン・アプリケーションからアクセスすることができます。プロセッサが制御アルゴリズムを実行し、PWM 変調器レジスタを更新できるようにするため、データがレディになると、ADCC は割込みを発生します。

図 17 に、代表的モーター・コントロール・アプリケーションでモーター電流帰還信号とその他のアナログ・モニタリング信号を取り込むための ADCC、CPU、SRAM、PWM、外部信号の相互接続を示します。この例では、エンコーダの正弦信号と余弦信号、ヒート・シンク温度、DC バス電圧をその他のモニタリング入力の例として示してあります。

信号帰還を正しく処理するための ADCC 設定の 3 つのステップを次に示します。

1. ADCC イベントの設定
2. 割込みとトリガのルーティング
3. データ・アクセスとメモリ割り当て

次のサブセクションでは、システムの正しいセットアップに必要とされる手順と関連レジスタの設定について説明します。

ADCC イベントの設定

図 17 に示す例に対する ADCC イベントの設定には、タイマー、ADC インターフェースとチャンネル、時間オフセット、同時サンプリング・スイッチに対する各イベントの割り当てが含まれています。これを行う方法は幾つかありますが、1 つの方法を図 16 と表 3 に示します。この例では説明目的のためにのみ両タイマーを使用しています。

この特定の例では、イベントを 1 つのタイマーにリンクすることができます。これはすべてのイベントの時間が PWM SYNC パルスを基準に決められるためです。両タイマーの使用が不可欠なケースは、2 軸モーター・コントロール・アルゴリズムのケースであり、2 セットの PWM 出力とそれらに対応する PWM 同期パルスを使います。

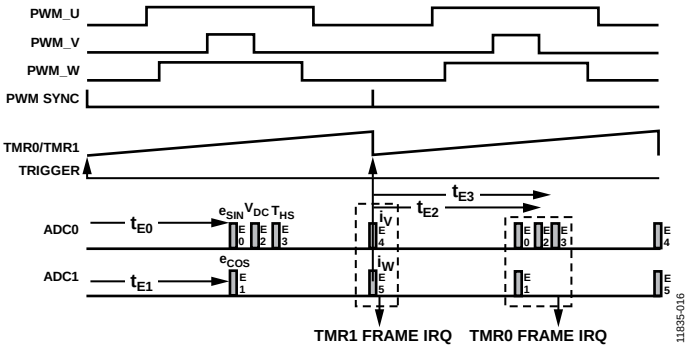

図 16.モーター・コントロール・アプリケーションでの代表的な ADCC の使用

表 3.アプリケーション例に対するイベント設定

Event	Timer	ADC I/F	ADC Ch	Time	Simultaneous Sample
E0 (e_{SIN})	TMR0	0	0	t_{E0}	Yes
E1 (e_{COS})	TMR0	1	0	$t_{E1} = t_{E0}$	Yes
E2 (V_{DC})	TMR0	0	2	t_{E2}	No
E3 (T_{HS})	TMR0	0	3	t_{E3}	No
E4 (i_V)	TMR1	0	1	0	Yes
E5 (i_W)	TMR1	1	1	0	Yes

相電流 i_V と i_W は、PWM 同期パルス・トリガが発生した直後に同時サンプリングされ、これらの相電流は TMR1 へリンクされます。タイマー1 フレームはメモリへ直ちに DMA 転送されて、新しい電流サンプルがメイン・アプリケーション・プログラムから使用可能になります。PWM サイクルの後半に、TMR0 にリンクされて、イベントの新しいフレームがサンプリングされます。エンコーダの正弦信号と余弦信号が同時にサンプリングされ、この後に DC バス電圧信号とヒート・シンク温度信号が続きます。3 つの ADC0 信号は、スループットを大きくする手段としてパイプライン化されます。次に TMR0 フレームがメモリへ DMA 転送されます。

これらのパラメータを設定するためには、イベント番号 nn ごとにイベント・コントロール・レジスタ ADCC_EVCTLnn とイベント時間レジスタ ADCC_EVTnn を設定する必要があります。このセクションで説明するドライバ API は、この処理を簡素化するために提供されています。

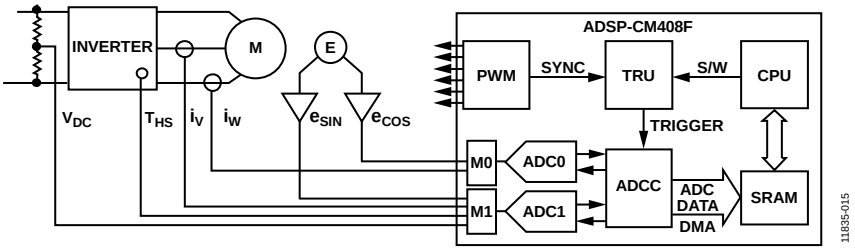

図 17.代表的なモーター・コントロール・アプリケーションでのシステム接続

割込みとトリガのルーティング

図 17 の例では、すべてのイベント時間は PWM サイクルを基準としているため、PWM 同期パルスで両タイマーがトリガされます。ADCC タイマーに対するハードウェア・トリガとしての PWM 同期パルスの接続では、まず ADCC トリガ・スレーブに対するマスター・トリガとして、TRU を PWM 同期パルスへ接続する設定が必要です。次に、ADCC タイマーを ADCC トリガにリンクさせる必要があります。

適切なトリガのルーティングの概念を図 18 に示します。マスター番号を該当するスレーブ・セレクト・レジスタ（この場合 TRU_SSR24）に書き込んで、トリガ・マスター19 （PWM0 SYNC）とトリガ・スレーブ 24 （ADCC_TRIG0）を接続します。次に、ADC_CTL レジスタの TRIGSEL ビットに該当する値を設定して、ADCC_TRIG0 トリガを 2 つのタイマーに接続します。

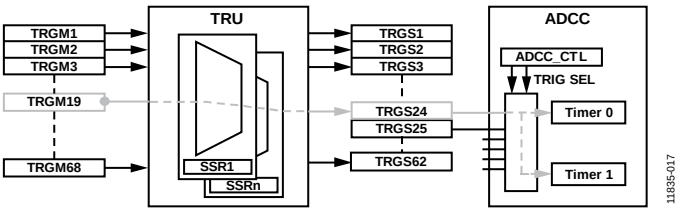

図 18. PWM 同期から ADCC タイマーへのトリガ・ルーティング

トリガ・ルーティングをこの構成にすると、PWM タイミングから ADC サンプリングまでのパスがハードウェアで直接リンクされるので、パス内でソフトウェア遅延は発生しません。また、トリガ・マスターには、GPIO ピン割込み、タイマー、カウンタ・イベントなど、その他のソースからもルーティングすることができます。この構成では、例えば ADSP-CM408F から制御されている他のコンバータなどとの正確な同期サンプリングが可能です。

さらに、ADCC タイマー・フレームの完了を他のペリフェラルまたはコア・スレーブに対するトリガ・マスターとして接続することができます。

この例では DMA 転送モードを使用しているため、ADCC_EIMSK レジスタを使ってすべてのイベント割込みをマスクする必要があります。ここでも、DMA モードで、フレーム割込みに対して該当する割込みサービス・ルーチンを登録するためのドライバ API が提供されています。

サンプル・タイミングを高精度化するトリガ・ルーティング

前述のように、現在のサンプリング・タイミングからチップ・セレクト・パルス幅遅れをなくすためには、トリガ・ルーティングの構成を少し変える必要があります。この場合、ADCC タイマーは GP タイマー・トリガからトリガされます。この GP タイマー自体は PWM 同期からトリガされます。このシーケンスを図 11 に示します。

データ・アクセスとメモリ割り当て

図 14 と図 15 に示すように、ADC データは、コア MMR 読出しにより、または DMA 転送で SRAM へ転送してアクセスすることができます。コア・モードでは、コア MMR 読出しを書き込む変数とは別のデータに対する特定のメモリ割り当ては不要です。ただし、DMA モードでは、特定のメモリ領域を割り当てて、DMA アクセス用に設定する必要があります。これを各タイマーに対して実行する必要があります。必要とされるメモリ・サイズは、各タイマーに接続されるフレーム・サイズと新しいフレームで上書きされる前にメモリへの格納が必要なフレーム数に依存します。

図 19 に、概念的な SRAM マップと SRAM 構成を制御する関連 ADCC レジスタを示します。ADCC_BPTR レジスタは、格納対象となる ADC サンプルのメモリ・ベース・アドレスに対するポインタを格納する必要があります。複数のフレームをメモリ・バッファに格納する必要がある場合、ADCC_FRINC レジスタは、次のフレームのベースに対するポインタのオフセットを格納します。ADCC_CBSIZ レジスタにゼロを書き込むと起動されるリニア・バッファ・モードでは、追加フレームはフレーム・インクリメント値の間隔で昇順に連続してメモリに格納されます。非ゼロ値 M を ADCC_CBSIZ に書き込むと、サーキュラ・バッファが起動され、フレーム・ベース・ポインタが ADCC_BPTR 値に戻り、既存フレームの上書きが開始される前に M フレームがメモリに書込まれます。

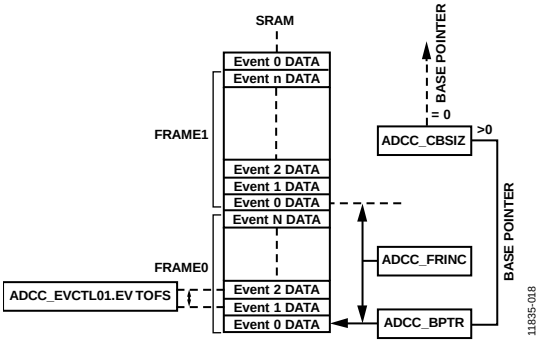

図 19. ADC DMA 転送用のメモリ構成

図 17 のモーター・コントロール・アプリケーション例では、各 PWM サイクルで ADC サンプルが集められて、コントロールおよびモニタリング・アプリケーション内で直ちに使用されます。そのため、サンプルを連続的に格納する意味はありません。メモリがすぐに満杯になるためです。このようなアプリケーションでは、M を 1 または小さい値に制限してサーキュラ・バッファをイネーブルするか、または ADCC_FRINC 値に 0 を設定して、PWM サイクルごとにフレームを上書きさせることができます。この作業を簡素化するドライバのアプリケーション・プログラミング・インターフェース（API）を ADCC ソフトウェアのサポートのセクションで説明します。

ADCCソフトウェアのサポート

アナログ・デバイセズの ADSP-CM40x EZKIT に添付されているイネーブルメント・ソフトウェア・パッケージには、このアプリケーション・ノートで説明した ADCC モジュールのセットアップを簡素化する多くの API 関数呼び出しが含まれています。これらの呼び出しは、種々のレジスタの正しい設定、および発生する必要のあるステータス・アクノリッジを監視します。

コード例

このアプリケーション・ノートのコード例では、図 17 に示すモーター・コントロール・アプリケーションの設定方法と使用方法をステップごとに説明しています。デバイス・ドライバにより負荷が増えますが、ADCC モジュール・レジスタの設定が大幅に簡素化されます。

コードの最初のセクションでは、ドライバの API 呼び出しで使用する多くのパラメータと構成定数を定義しています。

行 1～行 10 では、各タイマーのフレームと対応するデータ・バッファ・サイズを決めています。デバッグ用の安全対策として、サンプル・バッファ長の割り当てにファクタ 2 を使用しています。メモリへの ADC サンプルの転送はすべてハードウェアでトリガされるため（DMA も含む）、新しいバッファをドライバに提供して、ADC バッファ・ポインタをリセットする前に、ソフトウェア・ブレーク・ポイントを行 122 に挿入すると、メモリが上書きされます。余裕を持たせる手段として追加バッファを与えておけば、このデバッグ問題の発生を防止することができます。バッファ内のフレーム数は 1 に指定されていますが、これは新しいフレームが渡されるごとに API がメモリ・バッファを上書きすることを意味します。すなわち、メモリ割り当ては各タイマーの 1 フレームに対してのみ必要とされます。

行 11～行 16 では、各イベントのサンプリング時間を ACLK サイクル数で指定しています（表 3 参照）。SMP_TIME1、SMP_TIME2、SMP_TIME3 の間隔はわずか 1 ACLK サイクルであることに注意してください。このセットアップにより、ADC0 内でこれらのイベントがパイプライン化されます。

行 17～行 44 では、各 ADC チャンネルの制御ワード、6 個のサンプリング・イベントに対するチャンネル割り当て、データ・バッファ内の各イベントに対するアレイ・インデックスが指定されています。

行 45～行 59 では、ADC 動作に必要な変数と関数のプロトタイプが宣言されています。ADCC メモリ・バッファと ADCC タイマ・メモリ・バッファに対するメモリ割り当てサイズは、API により規定されているため変更できません。1 つの ADCC セットアップ関数、1 つの TRU セットアップ関数、2 つの ISR コールバック（各 ADCC タイマに 1 つ）が登録されています。

行 60～行 91 は、メイン ADCC 設定関数 SetupADC（）です。最初のステップでは、イベント番号、ADC 制御ワード、ADC タイマー、同時サンプリング、メモリ・オフセットをそれぞれイベントごとに含んでいる構造体であるイベント設定テーブルを設定します。

ADCC イベントの設定後、ADCC のインスタンス、およびそのインスタンスに対応する ADCC タイマーをオープンする必要があります。次に、各タイマー・フレーム割込みのコールバック関数名をドライバ（行 72～行 73）に登録する必要があります。この後、DMA モードをイネーブルし（行 74）、ADCC クロックとチップ・セレクトを設定しています（行 75～行 78）。

次に、タイマーを設定し、両方とも ADCC_TRIG0 入力をトリガ・ソースとします。SetupTRU（）関数内で PWM 同期パルス・トリガ・マスターに対するトリガ・スレーブとして ADCC_TRIG0 トリガを個別に接続します（行 92～行 97）。これを図 18 に示します。これらの関数呼び出しで使用するデータ一覧は、アナログ・デバイセズのイネーブルメント・ソフトウェア・パッケージ・ドライバ・ドキュメントに記載されています。

行 81 では、行 62 で定義された EventCFG 構造体が adi_adcc_ConfigEvent ドライバ関数に渡され、次に adi_adcc_SetEventMask ドライバ関数が必要に応じてイベントをイネーブルまたはマスクします。この場合、すべてのイベントがイネーブルされます。最大の ADC スループットを得るためには、行 83 のデュアル・ビット・データ・インターフェースをイネーブルすることが重要です。これは、ADC から ADC クロックの 8 サイクル以内に 16 ビット・データを転送できることを意味します（デュアル・ビット・インターフェースをイネーブルしない場合は、行 76 の N_{CK} および行 77 と行 78 の t_{CSCS} にそれぞれ 16 と 17 を設定する必要があることに注意してください）。次に、メモリをデータ・バッファに割り当て、データ・バッファを ADCC に提供して adi_adcc_SubmitBuffer 呼び出しを使ってこれに書込みます。adi_adcc_SubmitBuffer の API 呼び出しは DMA モードでのみ動作するため、DMA モードを設定した後に、この API を使う必要があります。行 105 でメイン・アプリケーションからこのドライバ関数が再度呼び出されて、アプリケーションがバッファからデータを取り出すと、バッファを ADCC 制御へ戻します。最後に、すべての設定が完了した後、タイマーと ADCC 自体のインスタンスをイネーブルする必要があります。

行 92～行 97 には、TRU のセットアップが含まれ、これには TRU のインスタンスのオープン、PWM 同期マスターから ADCC スレーブへのトリガのルーティング、TRU のイネーブルが含まれます。

前述のように、アプリケーション・レベルでの ADC データの処理は、タイマ・イベントと対応する DMA 転送の完了割込みの後の ADCC タイマ・コールバックにより処理されます。

行 98～行 127 では、コールバックの実行を示しています。バッファされたデータはバッファ内の対応するロケーションから取り出されて、該当するグローバル変数へ格納されます。この例では、更新された相電流データは、行 117 のアルゴリズム呼び出し MotorControl（）を使ってタイマー1 コールバックから呼び出されるモーター・コントロール・アルゴリズム内で直ちに使用されます。

ADCC イベント・タイマー割込みのサービスは、ADCC データのアクセス時に発生する単なるソフトウェア呼び出しであることに注意してください。同期とタイミングのすべては、ハードウェア・レベルで発生します。

行 128～行 136 では、TRU と ADCC のセットアップ関数に挿入して、図 11 に示す高精度サンプリング・タイミング機能を可能にする追加コード断片を提供しています。行 128～行 129 では、PWM 同期から GP タイマーTMR7 へ、さらに ADCC タイマー0 トリガへのハードウェア・トリガ・ルーティング・パスが設定されます。

行 130～行 136 では、ADC セットアップ関数に挿入して GP タイマーTMR7 を正しく設定／イネーブルして、正しい遅延を提供するサンプル・コードを提供しています。

すべてのケースで、SetupTRU 関数呼び出しは、SetupADC 関数呼び出しの前に行う必要があります。

```
/*********************************************
ADCC Module Setup Code Example
*********************************************/

/*******************Defines*********************/
1.  #define ADCC_DEVICE_NUM         0
2.  #define TRU_DEV_NUM             0
3.  #define ADI_TRU_REQ_MEMORY
4.  #define NUM_SAMPLES0       4
5.  #define NUM_SAMPLES1       2          /*
    Length of ADC buffers */
6.  #define FRAME_INC0
    2*NUM_SAMPLES0*sizeof(short)
7.  #define FRAME_INC1
    2*NUM_SAMPLES1*sizeof(short)  /* Frame
    increment in number of bytes for each buffer*/
8.  #define FRAMES_IN_BUFFER  1   /*Number of
    frames in buffer */
9.  #define NO_OF_EVENTS     6         /* Total
    number of events */
10. #define EVENT_MASK              0xFFFF

/*Event Times in ACLK Cycles*/
11. #define SMP_TIME0            950
12. #define SMP_TIME1            950
13. #define SMP_TIME2            951
14. #define SMP_TIME3            952
15. #define SMP_TIME4            0
16. #define SMP_TIME5            0

/* Control Words for All ADC Channels */
/*Upper Nibble = Chan No. Lower Nibble = 0xF for
    Sim Sampling, 0xD Otherwise*/
17. #define ADC0_VIN00_CTL          0x0F
18. #define ADC0_VIN01_CTL          0x1F
19. #define ADC0_VIN02_CTL          0x2D
20. #define ADC0_VIN03_CTL          0x3D
21. #define ADC0_VIN04_CTL          0x4D
22. #define ADC0_VIN05_CTL          0x5D
23. #define ADC0_VIN06_CTL          0x6D
24. #define ADC0_VIN07_CTL          0x7D

25. #define ADC1_VIN00_CTL          0x0F
26. #define ADC1_VIN01_CTL          0x1F
27. #define ADC1_VIN02_CTL          0x2D
28. #define ADC1_VIN03_CTL          0x3D
29. #define ADC1_VIN04_CTL          0x4D
30. #define ADC1_VIN05_CTL          0x5D
31. #define ADC1_VIN06_CTL          0x6D
32. #define ADC1_VIN07_CTL          0x7D

/*Mapping the Signals to the Appropriate ADC
    Channels*/
33. #define ES_CTL                 ADC0_VIN00_CTL
34. #define EC_CTL                 ADC1_VIN00_CTL
35. #define VDC_CTL                ADC0_VIN02_CTL
36. #define THS_CTL                ADC0_VIN03_CTL
37. #define IV_CTL                 ADC0_VIN01_CTL
38. #define IW_CTL                 ADC1_VIN01_CTL

/*Locations of ADC Signals in Data Buffer Index*/
39. #define IV_ADC                  0
40. #define IW_ADC                  1
41. #define ES_ADC                  0
42. #define EC_ADC                  1
43. #define VDC_ADC                 2
44. #define THS_ADC                 3

/******************Variables*******************/
45. static ADI_ADCC_HANDLE hADCC;             /*
    ADCC Handle */
46. static ADI_ADCC_HANDLE hADCCTimer0,
    hADCCTimer1;   /*ADCC Timer Handles*/
47. static uint8_t ADCCMemory[ADI_ADCC_MEMORY];
    /* Memory buffer for the ADCC device -
    predefined */
48. static uint8_t
    ADCCTmr0Memory[ADI_ADCC_TMR_MEMORY];
49. static uint8_t
    ADCCTmr1Memory[ADI_ADCC_TMR_MEMORY];    /*
    Memory buffer for the ADCC Timers -
    predefined*/
50. static uint16_t SampleBuffer0[NUM_SAMPLES0];
51. static uint16_t SampleBuffer1[NUM_SAMPLES1];
/* Memory buffer for the ADC samples */
52. static uint16_t Iv_adc, Iw_adc;
53. static uint16_t Es_adc, Ec_adc, Vdc_adc,
    Ths_adc;
/*Variables for ADC data*/
54. static uint8_t
    TruDevMemory[ADI_TRU_REQ_MEMORY];
55. static ADI_TRU_HANDLE hTru;
   /*TRU Device Memory and Handle*/

/*************Function Prototypes***************/
56. void SetupADC(void);
57. void SetupTRU(void);
58. static void AdccTmr0Callback(void *pCBParam,
    uint32_t Event, void *pArg);
59. static void AdccTmr1Callback(void *pCBParam,
    uint32_t Event, void *pArg);

/******Function to Configure ADCC**************/
60. void SetupADC(void) {
61. static ADI_ADCC_RESULT result;

/*Set Up Event Configuration Table*/
62. ADI_ADCC_EVENT_CFG  EventCFG[NO_OF_EVENTS] = {
63. {0,  ES_CTL, ADI_ADCC_ADCIF0, ADI_ADCC_TIMER0,
    true,  0, SMP_TIME0},
64. {1,  EC_CTL, ADI_ADCC_ADCIF1, ADI_ADCC_TIMER0,
    true,  2, SMP_TIME1},
65. {2,  VDC_CTL, ADI_ADCC_ADCIF0,
    ADI_ADCC_TIMER0, false,  4, SMP_TIME2 },
66. {3,  THS_CTL, ADI_ADCC_ADCIF0,
    ADI_ADCC_TIMER0, false,  6, SMP_TIME3 },
67. {4,  IV_CTL, ADI_ADCC_ADCIF0, ADI_ADCC_TIMER1,
    true,  8, SMP_TIME4 },
68. {5,  IW_CTL, ADI_ADCC_ADCIF1, ADI_ADCC_TIMER1,
    true, 10, SMP_TIME5 }}; /*Event#, CTL_WORD,
    ADC Interface, Timer ID, sim. samp, Mem offset
    in frame, Event time */
```

```
/*ADCC Setup API Functions*/
69. result = adi_adcc_OpenDevice(ADCC_DEVICE_NUM,
    ADCCMemory, &hADCC);
70. result = adi_adcc_OpenTimer(hADCC,
    ADI_ADCC_TIMER0, ADCCTmr0Memory,
    &hADCCTimer0);
71. result = adi_adcc_OpenTimer(hADCC,
    ADI_ADCC_TIMER1, ADCCTmr1Memory,
    &hADCCTimer1); /* ADCC Device handle, Timer to
    open, Timer memory, Pointer to the timer
    handle */
72. result = adi_adcc_RegisterTmrCallback
    (hADCCTimer0, AdccTmr0Callback, hADCCTimer0);
73. result = adi_adcc_RegisterTmrCallback
    (hADCCTimer1, AdccTmr1Callback,
    hADCCTimer1);/*Register callback functions*/
74. result = adi_adcc_EnableDMAMode(hADCC,true);

75. result = adi_adcc_ConfigADCCClock(hADCC,
    ADI_ADCC_ADCIF0, false,1u, 8u );
76. result = adi_adcc_ConfigADCCClock(hADCC,
    ADI_ADCC_ADCIF1, false,1u, 8u ); /*For each
    ADC interface: ADCC handle, ADC Interface
    number, falling edge, ACLK Clock divide, NCK*/
77. result = adi_adcc_ConfigChipSelect(hADCC,
    ADI_ADCC_ADCIF0, false, 1u, 0u, 9);
78. result = adi_adcc_ConfigChipSelect(hADCC,
    ADI_ADCC_ADCIF1, false, 1u, 0u, 9);/*For each
    interface: ADCC handle, ADC interface, active
    low, TCSCK, TCKCS, TCSCS*/
79. result = adi_adcc_ConfigTimer(hADCCTimer0,
    ADI_ADCC_TRIG0, true, false);
80. result = adi_adcc_ConfigTimer(hADCCTimer1,
    ADI_ADCC_TRIG0, true, false); /*For each
    timer: Timer handle, Timer trigger source,
    falling edge trigger, No trigger output */
81. result = adi_adcc_ConfigEvent(hADCC,
    &EventCFG[0], NO_OF_EVENTS);  /*ADCC handle,
    Pointer to the event configuration table,
    Number of events in the table */
82. result = adi_adcc_SetEventMask(hADCC,
    EVENT_MASK);                              /*
    Handle to the device, Enable all events */
83. adi_adcc_EnableDualBitDataIF(hADCC, true);
    /*Dual bit interface allows highest
    throughput*/
84. memset((void *)SampleBuffer0, 0, NUM_SAMPLES0
    * sizeof(short));
85. memset((void *) SampleBuffer1, 0, NUM_SAMPLES1
    * sizeof(short));
86. result = adi_adcc_SubmitBuffer(hADCCTimer0,
    SampleBuffer0, FRAME_INC0, FRAMES_IN_BUFFER);
87. result = adi_adcc_SubmitBuffer(hADCCTimer1,
    SampleBuffer1, FRAME_INC1, FRAMES_IN_BUFFER);
/*For each timer: timer handle, Pointer to the
    buffer, Frame increment,  Number of frames
    that fits into the given buffer */

88. result = adi_adcc_EnableTimer(hADCCTimer0,
    true);
89. result = adi_adcc_EnableTimer(hADCCTimer1,
    true);
90. result = adi_adcc_EnableDevice(hADCC, true);
    /*Enable everything*/
91. }
```

```
/*******Function to Configure TRU***************/
92. void SetupTRU(void){
93. ADI_TRU_RESULT result;
94. result = adi_tru_Open (TRU_DEV_NUM,
    &TruDevMemory[0], ADI_TRU_REQ_MEMORY, &hTru);
    /* Setup TRU for ADCC. Slave is ADCC0 trig 1
    and master is PWM0 SYNC pulse*/
95. result = adi_tru_TriggerRoute (hTru,
    TRGS_ADCC0_TRIG0, TRGM_PWM0_SYNC);  /*TRU
    device, slave, master*/
96. result = adi_tru_Enable (hTru, true); /*Enable
    TRU*/
97. }

/***********ADCC Timer Callbacks****************/
98. static void AdccTmr0Callback(void *pCBParam,
    uint32_t Event, void *pArg){
99.   switch(Event){
100.        case ADI_ADCC_EVENT_FRAME_PROCESSED:
101.          Es_adc= SampleBuffer0[ES_ADC];
102.          Ec_adc = SampleBuffer0[EC_ADC];
103.          Vdc_adc = SampleBuffer0[VDC_ADC];
104.          Ths_adc = SampleBuffer0[THS_ADC];
/*Store all of the data sampled in appropriate
    global variables*/
105.    _adcc_SubmitBuffer(hADCCTimer0,
    SampleBuffer0, FRAME_INC0, FRAMES_IN_BUFFER);
    /*Return the buffer to the ADCC for use in the
    next events*/
106.          break;
107.      case ADI_ADCC_EVENT_BUFFER_PROCESSED:
108.          break;
109.      default:
110.         break;
111.     }

112.   static void AdccTmr1Callback(void
    *pCBParam, uint32_t Event, void *pArg){

113.     switch(Event){
114.        case ADI_ADCC_EVENT_FRAME_PROCESSED:
115.          Iv_adc = SampleBuffer1[IV_ADC];
116.          Iw_adc = SampleBuffer1[IW_ADC];
117.          MotorControl();     /*Run the
    current control algorithm*/
118.
119.
120.          break;

121.      case ADI_ADCC_EVENT_BUFFER_PROCESSED:
122.   adi_adcc_SubmitBuffer(hADCCTimer1,
    SampleBuffer1, FRAME_INC1, FRAMES_IN_BUFFER);
123.          break;
124.      default:
125.         break;
126.     }
127.   return;
   }
```

```
/*********************************************
Enhanced Precision Timing Code
*********************************************/
/*Setup TRU for ADCC enhanced timing precision.
    Slave is ADCC0 trig 1 and master is GP timer 7
Added to SetpTRU() function in place of line 95 */

128.    result = adi_tru_TriggerRoute(hTru,
    TRGS_ADCC0_TRIG0, TRGM_TIMER0_TMR7);  // TRU
    device, slave, master
129.    result = adi_tru_TriggerRoute(hTru,
    TRGS_TIMER0_TMR7, TRGM_PWM0_SYNC);  // TRU
    device, slave, master

/*Setup GP timer 7 timer used to advance frame by
    one CS. Add to SetupADC() function after line
    91*/

130.    *pREG_TIMER0_STOP_CFG_SET =
    BITM_TIMER_STOP_CFG_TMR07;
131.    *pREG_TIMER0_RUN_CLR =
    BITM_TIMER_RUN_SET_TMR07; /*Disable Timer
    First*/
132.    *pREG_TIMER0_TMR7_CFG =
    ENUM_TIMER_TMR_CFG_PWMSING_MODE|ENUM_TIMER_TMR
    _CFG_IRQMODE1 |ENUM_TIMER_TMR_CFG_TRIGSTART |
    ENUM_TIMER_TMR_CFG_POS_EDGE|ENUM_TIMER_TMR_CFG
    _PADOUT_EN | ENUM_TIMER_TMR_CFG_EMU_CNT;
133.    *pREG_TIMER0_TMR7_DLY = (uint32_t)(fsysclk
    / F_SW - 0.00000045 * fsysclk); /* Delay must
    be Tsw minus one ADC chip-select. Chip select
    is 18 ACLKs*/
134.    *pREG_TIMER0_TMR7_WID = 16; /*Be careful
    here... DLY+WID must be smaller than one PWM
    period. In other words, WID must be smaller
    than one ADC chip select. If WID>CS, trigger
    pulse stretches into next PWM period. */
135.    *pREG_TIMER0_TRG_MSK &=
    ~(BITM_TIMER_TRG_MSK_TMR07);
136.    *pREG_TIMER0_TRG_IE |=
    BITM_TIMER_TRG_IE_TMR07; /*Enable TMR7*/
```

実験結果の例

コード例のセクションに示すコードの電流サンプリング部分を、クローズド・ループの永久磁石同期モーター・コントロール・アプリケーション回路でテストしました。アプリケーション回路は、図 4 の電流スケーリング・データを基準とする電流トランスデューサを使って、一般の AC ライン入力と−6.8 A～+6.8 A の制御されたモーター電流範囲で動作します。アプリケーション回路のサンプル結果を図 20～図 23 に示します。

図 20 に、1500 rpm の速度リファレンスと無負荷モーターで測定したモーター相電流を示します。モーター電流レベルは非常に小さく不連続です。

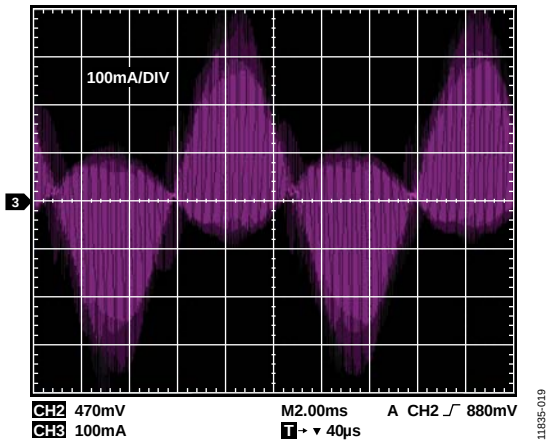

図 20.測定したモーター相電流

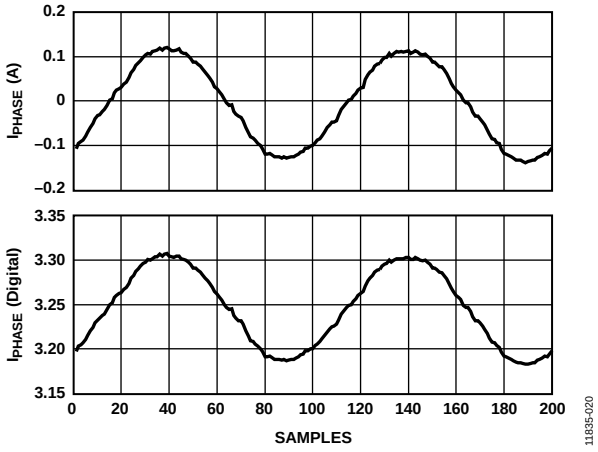

図 21. ADC でサンプリングしたモーター相電流
（上）スケーリングした実際の値と（下）デジタル・ワード出力

正しく同期化されたサンプリング方法の平均化処理効果を図 21 に示します。ここではモーター相電流の滑らかな正弦波に平均化された波形が最大 2%以下の電流レベルでも明確に見えます。図 21 および図 22 に、I_Q リファレンス電流のトラッキングでの制御ループの動作を示します。この図は、MATLAB® インターフェースへ RS-232 を経由して ADSP-CM408F から転送されたデータから作成しています。

図 23 では、PWM 同期パルスの位置、およびそれによるサンプリングのトリガを、電流が瞬時平均に一致する相電流 PWM サイクルの中心に示しています。このプロットは、分かり易くするため、比較的大きな負荷での特性を示しています。

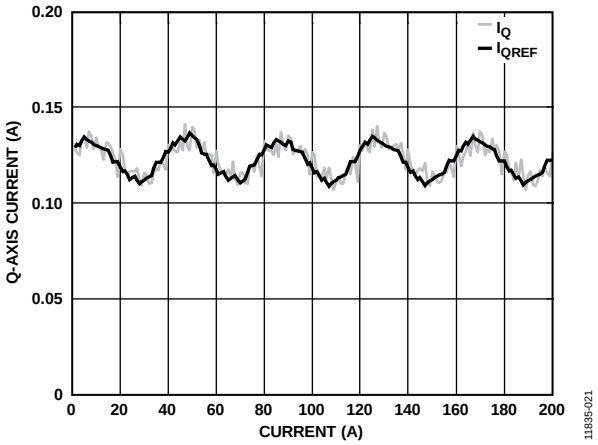

図 22. Q 軸のリファレンス電流と実際の電流

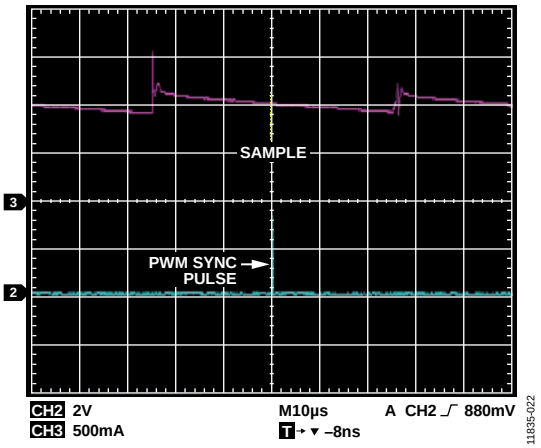

図 23.相電流とサンプリングの関係