AN-1267
APPLICATION NOTE

One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106, U.S.A. • Tel: 781.329.4700 • Fax: 781.461.3113 • www.analog.com

Motor Control Feedback Sample Timing Using the ADSP-CM408F ADC Controller

By Dara O'Sullivan, Jens Sorensen, and Aengus Murray

INTRODUCTION

This application note introduces the main features of the ADSP-CM408F analog-to-digital converter controller (ADCC) blocks with a focus on relevance and usefulness in current feedback systems of high performance motor control applications.

The purpose of this application note is to highlight the key capabilities of the analog-to-digital converter (ADC) module and to provide guidance on its configuration for motor control applications. Code samples illustrating the use of the ADCC drivers from Analog Devices, Inc., are provided.

Further details on the full range of features, configuration registers, and application program interfaces (APIs) for this ADCC may be found in the ADSP-CM40x Mixed-Signal Control Processor with ARM Cortex-M4 Hardware Reference Manual, and in the ADSP-CM40x Enablement Software Users Guide available on the ADSP-CM402F/ADSP-CM403F/ADSP-CM407F/ADSP-CM408F product pages.

While this application note is focused on current feedback, similar principles of configuration and application are applicable to the feedback and measurement of other signals.

Likewise, the focus of the application note is specifically on the ADSP-CM408F; however, the principles are generally applicable to the other parts within the ADSP-CM402F/ADSP-CM403F/ADSP-CM407F/ADSP-CM408F family.

TABLE OF CONTENTS

REVISION HISTORY

9/13—Revision 0: Initial Version

CURRENT FEEDBACK SYSTEM OVERVIEW

An example of current feedback in a motor control application is illustrated in Figure 1. This arrangement is typical of high performance motor drives in which motor phase winding currents are sampled rather than inverter low-side phase legs. At medium to high current levels, current transducers or transformers, CT0 and CT1, must be used in the current measurement path because resistive current shunts become too bulky and inefficient.

In this setup, the processor is located on the safe low voltage side of the isolation barrier, with signal isolation usually being inherent to CT0 and CT1, and with digital isolation also existing between the PWM outputs of the microprocessor and the gate drivers.

Generally, some signal conditioning is required between the outputs of the current transducers and the inputs to the ADC for range matching and high frequency noise filtering. The conditioned current measurement signals are then applied to the ADC inputs for sampling and conversion. Applying one winding current measurement to each of the ADC inputs enables simultaneous sampling of the current measurements for greater control loop accuracy, and consequent performance enhancement. Furthermore, synchronization of the sampling instant with the PWM SYNC pulse is also configurable directly in hardware.

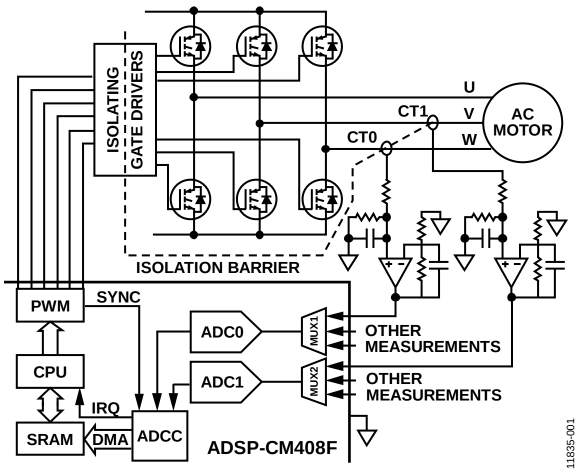

Figure 1. Current Feedback to ADSP-CM408F ADC in Motor Control

These features enable precise timing of the point in the PWM cycle at which the phase currents are measured. Aligning this measurement instant with the midpoint of the zero vector or the midpoint of the PWM cycle ensures that the level at which the current is being sampled is effectively equal to the instantaneous average current with switching ripple being ignored.

This is depicted in Figure 2 in which simultaneous U-phase and V-phase sampling is shown occurring at both zero vector midpoint and PWM cycle midpoint.

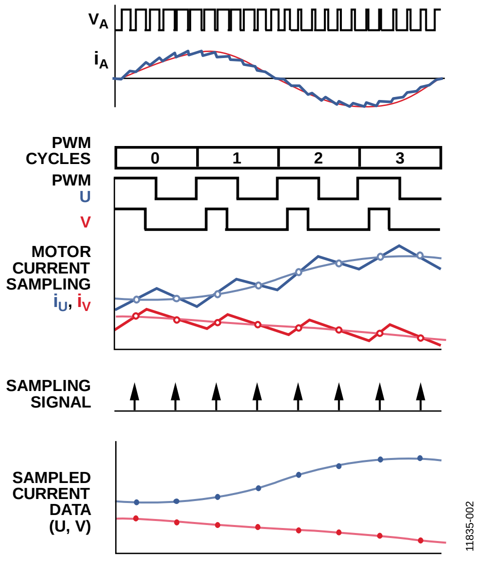

Figure 2. Illustration of Average Current Sampling

Once conversion of the data is complete, it can be transferred via direct memory access (DMA) to the controller SRAM; an interrupt is generated on completion of the transfer. Direct ADC status and data reads are also possible in core mode through memory mapped registers, but this method involves more processor overhead.

Typically, other analog signals, such as dc bus voltage, IGBT temperature, and motor position sine and cosine outputs, are also sampled. Though this application note focuses on current feedback, much of the information is also relevant to other measurement parameters within the system.

ADC MODULE OVERVIEW

The ADC is a dual, 16-bit, high speed, low power, successive approximation register (SAR) design with up to 14 bits of accuracy.

The input multiplexers enable up to a combined 26 analog input sources to the two independently controlled ADCs (12 analog inputs plus one DAC loopback input per ADC) with two channels simultaneously sampled at any given time. ADC conversion times are as fast as 380 ns. The voltage input range requirement for the single-ended analog inputs is from 0 V to 2.5 V.

An on-chip buffer between the multiplexer and ADC reduces the need for additional signal conditioning external to the ADSP-CM408F. Additionally, each ADC has an on-chip 2.5 V reference that can be overdriven when an external voltage reference is preferred (and by selecting this option using the ADCC_CFG register).

A graphical overview of the overall analog subsystem within the ADSP-CM408F is shown in Figure 3. The ADSP-CM408F is a multiple die system-in-package (SiP) and the ADC silicon is manufactured on a different process than the processor silicon as shown in Figure 3.

The ADCC is responsible for synchronizing timing within the ADC with the processor, and for managing direct memory access (DMA) transfers of sampled data to SRAM.

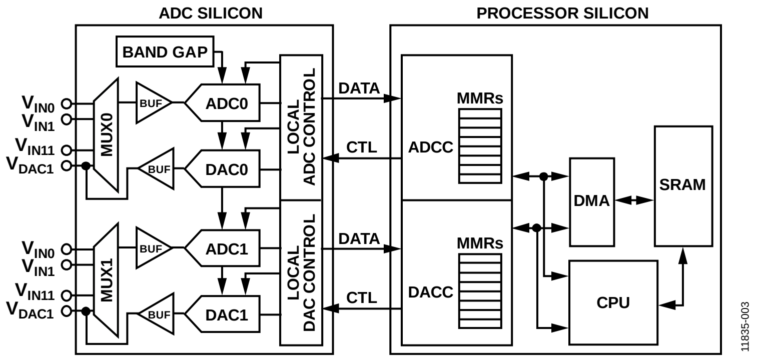

Figure 3. ADSP-CM408F Analog Subsystem

CURRENT FEEDBACK SCALING

To correctly utilize the ADC capability over the maximum range, it is important to scale the feedback signals in the correct manner. The signal progress through the feedback path is illustrated in Figure 5. The bipolar phase winding current, I_W, is converted to a unipolar voltage presented at the input of the ADC by the combined functionality of the current transducer (or transformer) and signal conditioning circuitry.

The transfer function of the current transducer is represented by the equation

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

where:
V_{IW} is the output voltage.
K_{CT} is the linear gain coefficient of the transducer.
V_{0CT} is the zero current offset voltage of the transducer.

In reality, K_{CT} tends to be nonlinear at some current levels in various transducer types, and for increased accuracy should be expressed as a function of I_W, that is, K_{CT} (I_W). The ADC input voltage is then expressed as

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

where K_{SIG} is the low frequency gain of the signal conditioning circuitry.

This unipolar voltage is converted to a 16-bit unsigned integer which is DMA transferred to the processor memory, after which an interrupt alerts the control program that a new data sample is available. The idealized transfer function of the ADC is given by

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

where:
N_{IW} is the ADC digital output word.
K_{ADC} represents the linear gain of the ADC and is equal to the ADC resolution divided by the input voltage range as indicated.

There will be some offset associated with the output of the ADC and within the software it is generally a good approach to include some offset compensation, N_{ADC_OFFSET}, which subtracts from the ADC output to take account of any offset within the ADC itself, plus any residual offset from the transducer and signal conditioning. This value can be dynamically updated during periods of zero current, such as system startup or disabled drive output.

Finally, the digital representation of the current transducer zero current offset voltage, N_{CT_OFFSET}, is subtracted from the ADC output to give the signed value IW which is related to the actual phase winding current as

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

where

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

This signed 16-bit value can be converted to a floating point value or used directly depending on the controller implementation. For optimum use of the full ADC range, the peak positive controlled current in the system should correspond to an ADC input voltage of 2.5 V, with peak negative controlled current corresponding to an ADC input
of 0 V.

An example of this is shown in Figure 4, which depicts a typical current waveform and the various zero, peak, and nominal levels associated with it and how they propagate through the signal measurement system of Figure 5.

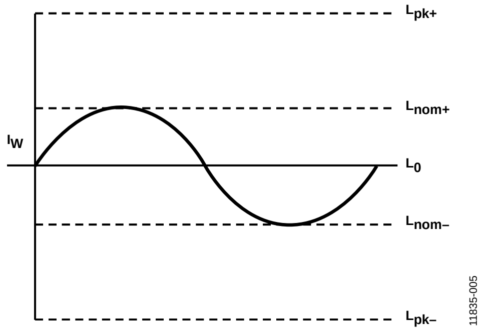

Figure 4. Current Feedback Signal Amplitudes

Table 1. Current Feedback Signal Amplitudes

Level	I_W	V_{IW}	V_{IW_ADC}	N_{IW}
L_{pk+}	6.8 A	4.625 V	2.313 V	0xECD9
L_{nom+}	4 A	3.75 V	1.875 V	0xC000
L_0	0 A	2.5 V	1.25 V	0x8000
L_{nom-}	−4 A	+1.25 V	+0.625 V	0x4000
L_{pk-}	−6.8 A	+0.375 V	+0.188 V	0x1340

This example is based on a CAS 6-NP Hall effect current transducer from LEM, with three primary turns giving a 0 V to 5 V output, followed by signal conditioning circuitry with a gain of 0.5.

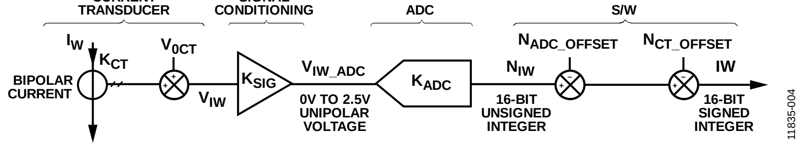

Figure 5. Scaling Relationships in Current Feedback Path

ADC TIMING CONSIDERATIONS

Synchronization of the sampling events with the PWM cycle is important for accurate current feedback. The conceptual sequencing of ADCC operation with respect to the PWM cycle is illustrated in Figure 6. The following sequence of events is triggered by the PWM synchronization pulse.

1. The PWM SYNC pulse triggers the timer to start.
2. The ADCC continuously compares the sample time from the event information with the timer time.
3. A timer match occurs and the ADCC schedules ADC operation.
4. Once the ADC is available, the appropriate channel is selected by the ADCC using the event information.
5. The ADCC triggers an ADC conversion sequence and the ADC samples and converts the data.
6. Data is streamed back to the ADCC.
7. Data is transferred by the ADCC to a memory location via DMA (LSB first).
8. An interrupt is generated that alerts the CPU that a data sample is available.

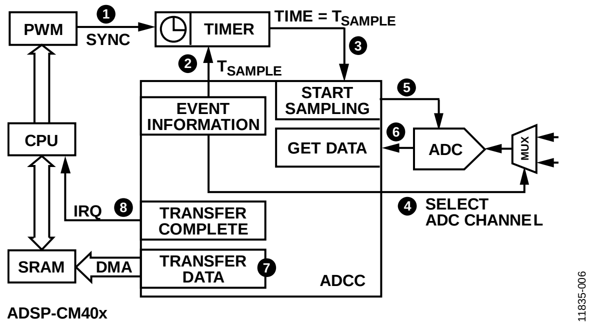

Figure 6. ADCC Operation Sequence

The sections that follow describe this event sequence and its implementation within the ADCC. In the event where multiple ADC sampling requests overlap, the ADCC implements a scheduling scheme known as pipelining, which is also described in detail.

ADCC EVENT TIMING

The controller manages the configuration and timing of up to 24 sampling events. The timing of these events is constrained by a trigger, which starts one of two timers (TMR0 or TMR1) and an event time after the timer starts.

As illustrated in the functional diagram of Figure 7, the trigger source can be selected from a range of peripheral or processor events, such as PWM SYNC pulses, timers, or I/O pin interrupts. Each event is associated with an event number depicted as Eventx, an event time, TIMEx, control information shown as CTLx, and its resultant data. The event control information, depicted as CTLx in Figure 7, contains information for each sample event, such as the ADC interface and channel numbers, the ADC timer being used, simultaneous sampling selection, and memory offset for the ADC data associated with the event. This information is used by the ADCC to multiplex the correct ADC channel, CHx, initiate ADC conversion (cvst0/cvst1 signals), and transfer the correct data to the appropriate event data register.

A DMA transfer can then be set up to move the ADC data for each event into SRAM. Upon completion of all of the events and subsequent DMA transfer, an interrupt is generated to inform the main application code that new ADC data is available.

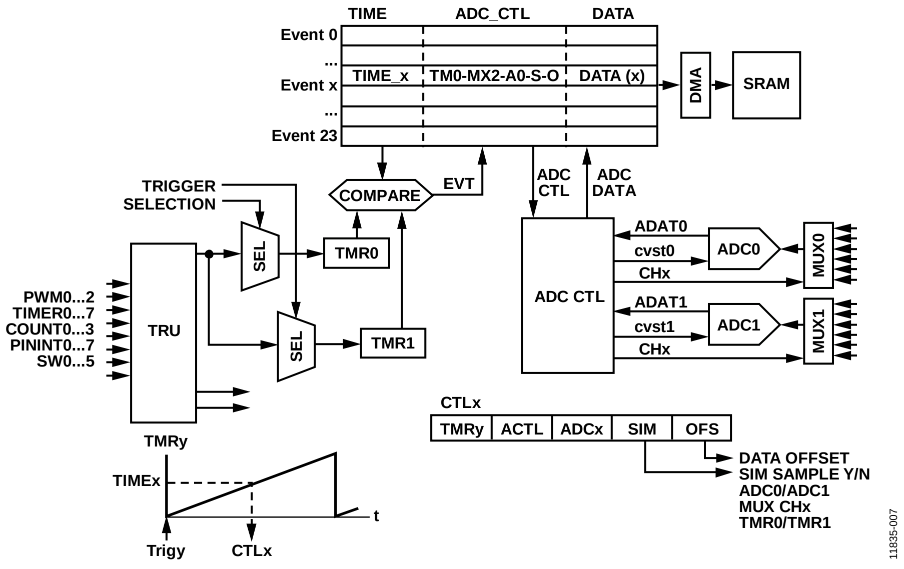

Figure 7. ADCC Module Functional Diagram

For example, Figure 8 depicts three sampling events associated with ADC Timer 0. The PWM SYNC pulse is the trigger for the timer, and event times are associated with each event. Clearly, Event 0 and Event 1 are simultaneous sampling events with the event time in the event time registers set to zero. Event 2 occurs at a later time, again, as determined by the time in the Event 2 time register, expressed in multiples of the ADC clock period, t_{ACLK}. In practice, if Event 2 is the final event associated with Timer 0, the timer stops running after the event has been handled to save power.

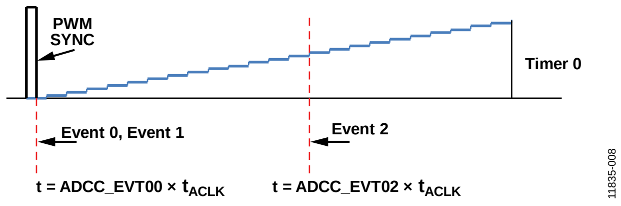

Figure 8. Event Timing

ADC OPERATIONAL TIMING

After a sampling event has been triggered by the ADCC controller, there is a conversion time latency associated with the ADC operation itself. This is shown in Figure 9 for a situation in which a single ADC event is associated with each ADC interface and simultaneous sampling of the two events is enabled.

Three discrete conversion cycles are associated with the ADC operation.

1. Writing the 8-bit control word that selects the ADC channel to be read (ADCC_EVTCTL.CTLWD).
2. Asserting the conversion pulse that enables ADC sampling and conversion.
3. Streaming the 16-bit ADC data back to the ADC.

The ADCC provides the chip select and gated clock signals for these three event phases. The ADCC interface to the ADC is a serial interface with a dual bit option. Therefore, the minimum number of clock cycles to be provided during each CS pulse (ADCC timing control register field NCK) is 8. Other important settings are the ADC clock frequency, the minimum delay (in ACLK cycles) between the conversion cycle chip selects (TCSCS), and the minimum delays between CS edges and ACLK edges (TCSCK and TCKCS). Consequently, the ADC conversion cycle time, t_{conv_adc}, for a single pair of simultaneously sampled signals is given by

$$t_{conv\_adc} = \frac{3}{f_{ACLK}}(TCSCK + NCK + TCKCS + TCSCS)$$

where f_{ACLK} is the frequency of the ADCC clock.

This is internally generated from the processor system clock, f_{SYSCLK}, by means of the divisor ACKDIV (in timing control Register ADCC_TCA) and is calculated as

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

The system clock is, in turn, derived from the processor core clock, $f_{CORECLK}$. Optimum system performance is achieved when $f_{CORECLK}$ is an integer multiple of f_{SYSCLK}. Upon completion of the ADC conversion, additional latency is associated with the DMA transfer of the ADC data to data memory and finally the servicing of the interrupt request that makes the data frame available to the main application program. Thus, the total time from trigger (for example, PWM SYNC pulse) to data availability in the application is equal to

$$t_{conv\_total} = t_{conv\_adc} + t_{DMA} + t_{IRQ}$$

where:

t_{DMA} is the average time for DMA transfer.

t_{IRQ} is the average time for interrupt request servicing.

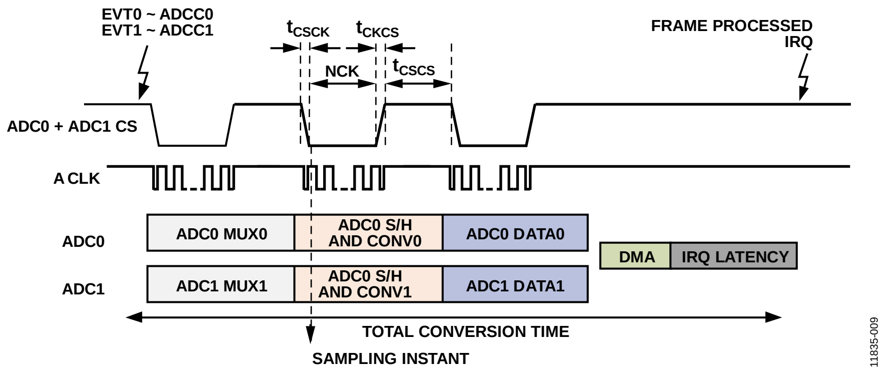

Figure 9. Conversion Timing of a Simultaneous Sampling Single Event

Typical timing settings are listed in Table 2. Some of the constraints on the times are also given. An absolute constraint for achieving correct performance of the ADC is that the total conversion time allowed, t_{conv_adc}, must be at least 380 ns. Of this period, the time between chip selects, t_{CSCS}—which corresponds to the acquisition time of the ADC—must be at least 150 ns. The resultant timings for a single simultaneous sampling event are outlined in Figure 10 relative to the sampling of the motor winding current (note that this figure is exaggerated for purposes of illustration).

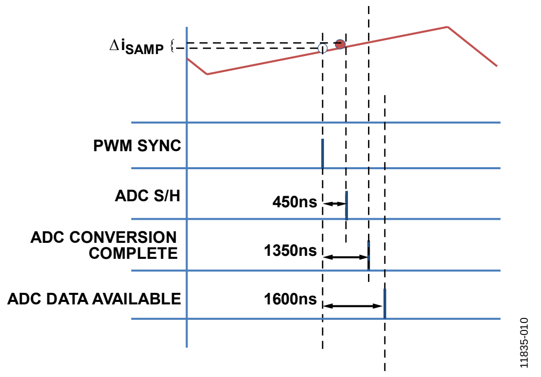

Figure 10. Sample Delay Times

Table 2. Timing Settings for a Typical ADC Setup

Quantity	Value	Comment	Set By
$f_{CORECLK}$	240 MHz	Maximum allowed	PLL configuration
f_{SYSCLK}	80 MHz	Maximum is 100 MHz	$= f_{CORECLK}/3$
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	

It is evident that with these settings there is an offset of 450 ns between the desired sampling point on the current waveform, and the actual point sampled. This is equal to one chip select pulse width (200 ns + 25 ns + 0 ns) plus one pulse width between chip selects (225 ns). This results in a difference of Δi_{samp} between the average motor winding current and the actual sampled current which may need to be accounted for in sample timing scheduling, although in the context of a typical current control loop bandwidth of 1 kHz, this represents <0.2° of phase shift. Moreover, for a typical PWM frequency of 10 kHz, ADC data is available to the application program within <2% of the available PWM cycle time from occurrence of the PWM SYNC pulse for the settings in Table 2. An additional latency of 4 to 5 SYSCLK cycles occurs between an event becoming active and the beginning of ADC operation if the ADC is in an idle state on occurrence of the event.

ADC PIPELINING

In the case where new events begin to overlap existing events that are being handled by the ADC, the ADCC stores the new events as pending events in an eight-deep FIFO buffer, one of which is available for each ADC interface. Once the control word has been written for an active event, the ADCC immediately initiates writing of the control word for the first pending event, while the active event sampling phase is occurring. Likewise, a second pending event has its control word phase initiated on completion of the control word phase for the first pending event. In this manner, the ADCC can interleave three parallel events together on each ADC interface in a pipelined manner. Thus, events can be spaced together in a compact and efficient manner.

Configuration of the event timing to achieve this pipelining of events results in the highest ADC throughput. This is illustrated in Figure 12 in which three pairs of simultaneous sampled events are triggered very close to each other. The ADCC begins to process Event 0 and Event 1, while storing Event 2 through Event 5 in the FIFOs. Subsequently, these events are handled as ADC resources become available.

It is evident from Figure 7 that during one of the CS assertions, the ADCC is handling all six events at varying stages of each event, and that the time spacing between consecutive samples is only equal to 18 ACLK cycles. This corresponds to 450 ns for the settings of Table 2 and can clearly be reduced further by increasing the ACLK frequency. To maximize the bandwidth of the ADC within the motor control application, the best approach is to deliberately pipeline all of the PWM cycle related sampling events. This ensures that new ADC samples are available at the earliest opportunity within the PWM cycle. This implies that all events should have their event times very close to zero, that is, immediately after the PWM SYNC pulse. However, nonsimultaneous sampled events linked to the same timer and ADC interface should not have the same event time.

It is recommended to allow a minimum of 1 ACLK cycle between the event times stored in the event time registers, ADCC_EVTnn (nn is the number of registers from 0 to 24), to allow for correct scheduling. With pipelining operational, the total conversion time including start-up latency, DMA transfer, and interrupt servicing is shown in Figure 11 for different simultaneously sampled pair numbers with the timing settings of Table 2.

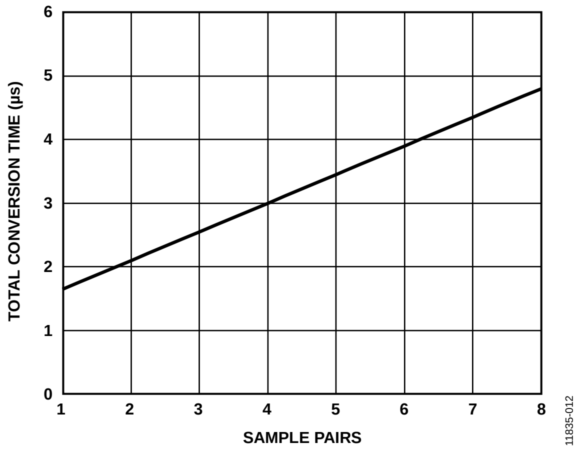

Figure 11. Total Conversion Time for Different Sample Pair Numbers

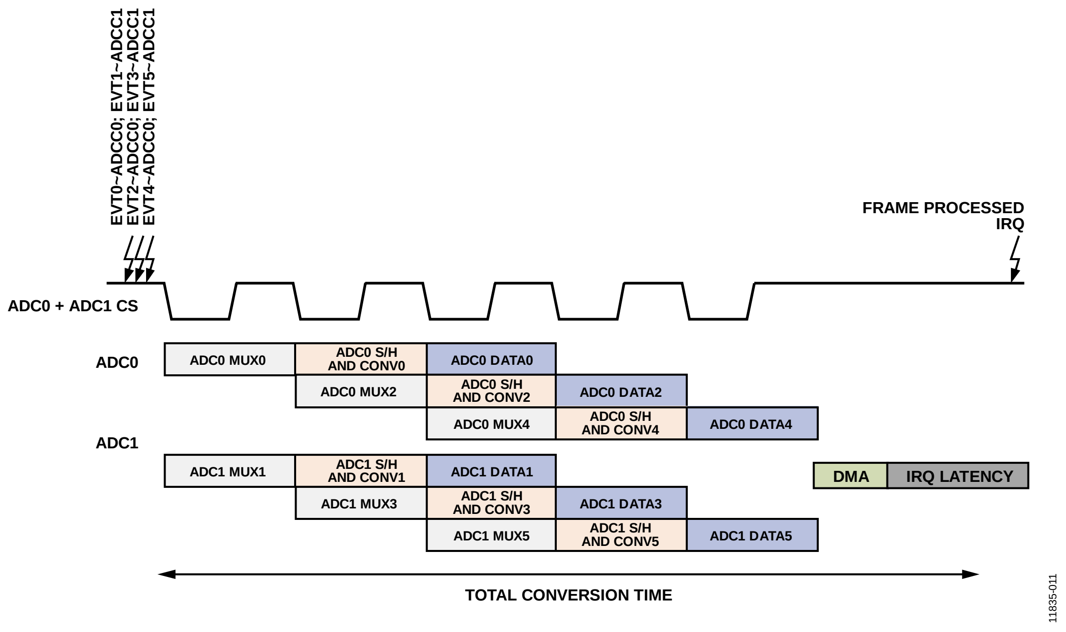

Figure 12. Pipelining of Events Within the ADC

ADC DATA ACCESS

The examples shown thus far have all assumed that the ADC data is accessed in memory via automatic DMA transfer. Data access directly from core reads of the ADCC memory mapped registers (MMRs) is also possible as shown in Figure 13. Note that the label ACK in Figure 13 represents an acknowledge signal, not the analog clock.

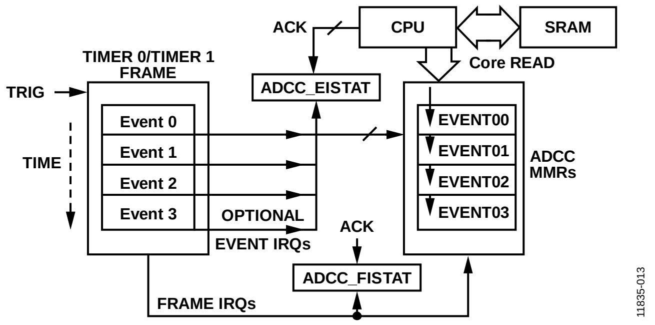

Figure 13. ADC Data Access in Core Mode

In core mode, the CPU is signaled regarding the readiness of new data via either event or frame interrupts, which can be individually masked or unmasked as desired. The additional flexibility in this mode is that individual events can be read as soon as they are completed, before the entire frame of events has completed. The disadvantage of core mode is that the overall latency involved in the interrupt servicing and MMR read accesses is higher than in DMA mode. With optimal core and clock ratio settings, each MMR read takes 10 to 12 SYSCLK cycles on top of the latency associated with each interrupt service.

Data access in DMA mode is depicted in Figure 14. In this case, DMA transfers only take place after the completion of a timer frame, and the frame interrupt signals the CPU only after the DMA transfer has been completed.

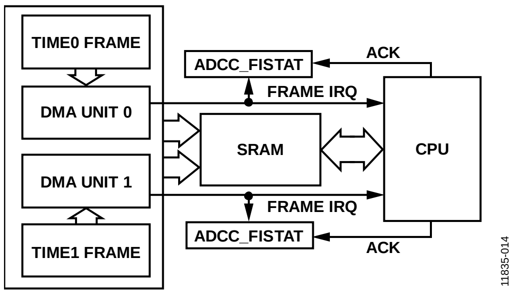

Figure 14. ADC Data Access in DMA Mode

In both cases, the EISTAT and FISTAT registers provide status indications of the event and frame interrupts, where these are active, and these must be acknowledged by the CPU by clearing the relevant bits before the next trigger occurrence, or a trigger overrun condition will be flagged.

ADCC DATA FAULT DETECTION

The ADCC has a number of error status register bits that are set on occurrence of data faults that can occur due to incorrect setup of the ADCC event timings, and/or nondeterministic event sequences. These faults can overload the ADCC or result in invalid ADC data and comprise the following:

- Trigger overrun. Next trigger occurs before current frame has completed.
- DMA bandwidth. Frame completion is taking longer than user defined time.
- Memory error. Unsuccessful ADC data write.
- Event collision. A new event occurs while processing an existing event.
- Event miss. An event does not get processed.

All of these errors are configurable as interrupt sources to the core, if desired, and all of them set bits in the ADCC_ERRSTAT register. In a motor control context, and particularly for current feedback measurements, errors related to event miss, memory, and trigger overruns are important to monitor in the core application because incorrect or missing current loop data could result in control loop instability. Event collision is a normal occurrence in pipelined operation and is not generally critical unless the FIFO becomes full.

ADCC MODULE, TRIGGER ROUTING, AND MEMORY SETUP

There are a number of steps in setting up the ADCC module along with the trigger routing unit and data buffers before the ADC is ready for use. Once configured, assuming DMA data access mode, the DMA engine automatically streams primary ADC data to memory where it can be accessed from within the main application. The ADCC generates an interrupt when data is ready so the processor can execute the control algorithm and update the PWM modulator registers.

Figure 15 outlines the interconnections required between the ADCC, CPU, SRAM, PWM, and external signals to capture motor current feedback and other analog monitoring signals in a typical motor control application. In this example, encoder sine/cosine signals, heat sink temperature, and dc bus voltage are provided as examples of additional monitoring input.

The three steps for setting up the ADCC to correctly handle the signal feedback are as follows:

1. ADCC event configuration.
2. Interrupt and trigger routing.
3. Data access and memory allocation.

The following subsections describe the procedure and the relevant register configurations required for correct setup of the system.

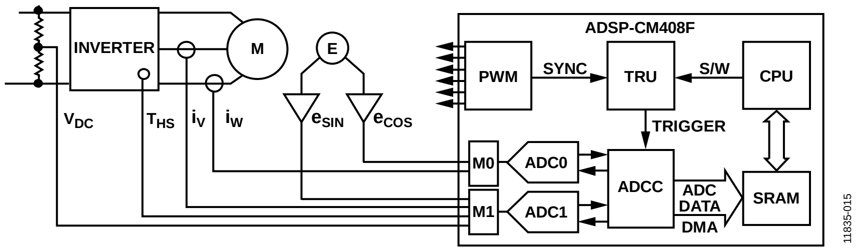

Figure 15. System Interconnections in Typical Motor Control Application

CONFIGURATION OF ADCC EVENTS

Configuration of the ADCC events for the example shown in Figure 15 involves assignment of each event with a timer, an ADC interface and channel, a time offset, and a simultaneous sampling switch. This can be achieved in several ways, but one possibility is shown in Figure 16, and listed in Table 3. This example utilizes both timers for illustration purposes only.

In reality, for this specific example, the events could be linked to one timer because all of the events are timed in relation to the PWM SYNC pulse. A use case in which the use of both timers would be essential is a dual axis motor control algorithm, which uses two sets of PWM outputs and their corresponding PWM SYNC pulses.

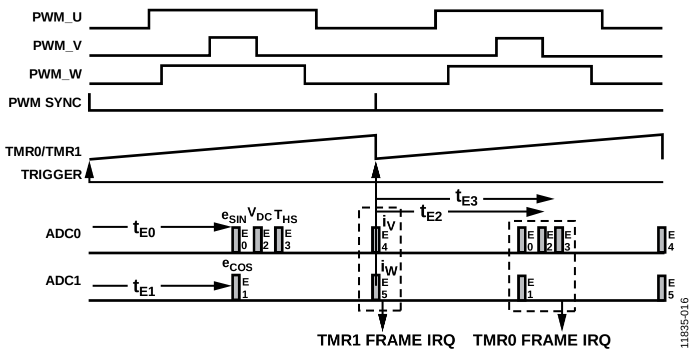

Figure 16. Typical ADCC use in a Motor Control Application

Table 3. Event Configurations for Example Application

Event	Timer	ADC I/F	ADC Ch	Time	Simultaneous Sample
E0 (e_{sin})	TMR0	0	0	t_{E0}	Y
E1 (e_{cos})	TMR0	1	0	$t_{E1} = t_{E0}$	Y
E2 (V_{DC})	TMR0	0	2	t_{E2}	N
E3 (T_{HS})	TMR0	0	3	t_{E3}	N
E4 (i_V)	TMR1	0	1	0	Y
E5 (i_W)	TMR1	1	1	0	Y

Phase currents i_V and i_W are simultaneously sampled immediately after the PWM SYNC pulse trigger has occurred and these are linked to TMR1. The Timer 1 frame is immediately DMA transferred to memory and the new current samples are available to be used by the main application program. At a later point in the PWM cycle, linked to TMR0, a new frame of events is sampled. The encoder sine-cosine signals are simultaneously sampled, closely followed by the dc bus voltage and heat sink temperature signals. The three ADC0 signals are pipelined for maximum throughput. The TMR0 frame is then DMA transferred to memory.

Configuration of these parameters requires programming of the ADCC_EVCTLnn event control register and ADCC_EVTnn and event time register for each event number nn. Driver APIs described in this section are available to simplify this process.

INTERRUPTS AND TRIGGER ROUTING

In the example in Figure 15, all events in time are referenced to the PWM cycle and so both timers are triggered by the PWM SYNC pulse. The connection of the PWM SYNC pulse as a hardware trigger to the ADCC timers first requires configuration of the TRU to connect the PWM SYNC pulse as a master trigger to an ADCC trigger slave. Then, the ADCC timers must be linked to the ADCC trigger.

This is shown conceptually in Figure 17 and involves connection of the Trigger Master 19 (PWM0 SYNC) to Trigger Slave 24 (ADCC_TRIG0) by writing the master number in the appropriate slave select register, TRU_SSR24 in this case. The ADCC_TRIG0 trigger is then routed to the two timers by means of the TRIGSEL bits in the ADC_CTL register.

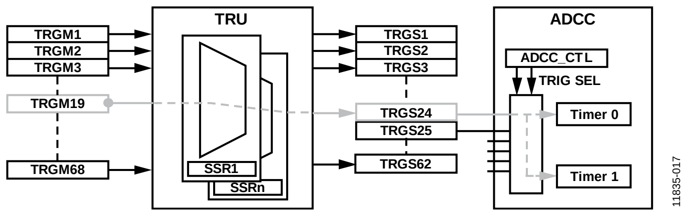

Figure 17. Trigger Routing from PWM SYNC to ADCC Timers

This provides a direct link in hardware from the PWM timing to the ADC sampling with no software latencies in the path. The trigger master can also be routed from other sources, such as GPIO pin interrupts, timer, counter events. This enables accurate synchronization of sampling with, for instance, other converters being controlled by the ADSP-CM408F.

Furthermore, completion of ADCC timer frames can be connected as trigger masters to other peripheral or core slaves.

Because DMA transfer mode is used in this example, all event interrupts should be masked in the ADCC_EIMSK register. Again, driver APIs are provided to register the appropriate interrupt service routines for the frame interrupts in DMA mode.

DATA ACCESS AND MEMORY ALLOCATION

As illustrated in Figure 13 and Figure 14, the ADC data can be accessed either via core MMR reads or by making it available in SRAM by DMA transfer. Clearly, in core mode, no specific memory allocation needs to be configured for the data apart from the variables to which the core MMR reads are being written. However in DMA mode, specific memory area must be allocated and then configured for the DMA access and this must be performed for each timer. The memory size required depends on the size of the frame associated with each timer, and on how many frames need to be stored in memory before being overwritten by new frames.

Figure 18 shows a conceptual SRAM map along with relevant ADCC registers that control the configuration of the SRAM. The ADCC_BPTR register must store a pointer to the memory base address for ADC samples to be stored. If more than one frame needs to be stored in the memory buffer, then the ADCC_FRINC register contains the offset for the pointer to the base of the next frame. In linear buffering mode—activated by writing zero to the ADCC_CBSIZ register—additional frames are stored in memory in a continuously increasing linear manner, spaced by the frame increment value. If a nonzero value, M, is written to ADCC_CBSIZ, then circular buffering is activated, and M frames are written to memory before the frame base pointer returns to the ADCC_BPTR value and begins to overwrite the existing frames.

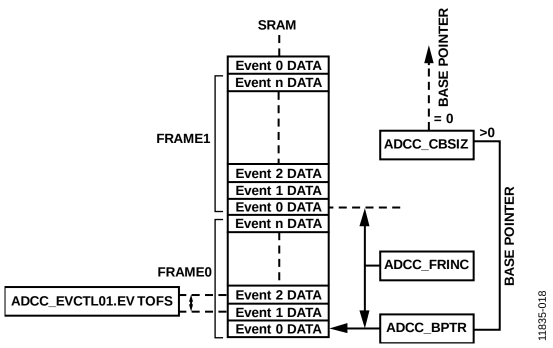

Figure 18. Memory Configuration for ADC DMA Transfers

In the motor control application example in Figure 15, the ADC samples are gathered every PWM cycle and are used immediately within the control and monitoring application. Therefore, it does not make sense to store them in a linear manner because memory would very quickly be overloaded. This can be achieved either by enabling circular buffering with M limited to 1 or some small value, or by simply setting the ADCC_FRINC value to zero, and overwriting the frame every PWM cycle. The driver APIs that simplify this task are outlined in the ADCC Software Support section.

ADCC SOFTWARE SUPPORT

The Analog Devices Enablement Software package provided with the ADSP-CM40x EZkit contains a number of API function calls that simplify the setup of the ADCC module discussed in this application note. These calls monitor correct configuration of the various register as well as any status acknowledgments that need to take place and so on.

EXAMPLE CODE

The example code in this application note illustrates a step-by-step approach to configuring and using the motor control application shown in Figure 15. The device driver adds some overhead, but significantly simplifies the programming of the ADCC module registers.

The first section of code defines a number of parameter and configuration constants used in the driver API calls.

Lines 2 through 7 define the frame and associated data buffer sizes for each timer. The number of frames in the buffer is defined as 1. This means that the API overwrites the memory buffer every time a new frame is submitted to it, that is, memory allocation is required for only one frame for each timer.

Lines 9 through 14 define the sample times for each event in ACLK cycle numbers as per Table 3. Note the separation of SMP_TIME1, SMP_TIME2, and SMP_TIME3 by only one ACLK cycle. This setup causes these events to be pipelined within ADC0.

Lines 17 to 44 define the control words for each ADC channel, the channel mapping for the six sampling events, and the array indices for each event within its data buffer.

Lines 45 to 59 declare the variables and function prototypes required for the ADC operation. The memory allocation sizes for the ADCC memory buffer and ADCCTmrx memory buffers are predefined by the API and should not be changed. One ADCC setup function, one TRU setup function, and two ISR callbacks—one for each ADCC timer—are registered.

Lines 60 to 91 contain the main ADCC configuration function SetupADC(). The first step is to set up the event configuration table, a *struct* which contains the event number, ADC control word, ADC timer, simultaneous sampling, and memory offset for each event.

Following successful configuration of the ADCC events, an instance of the ADCC must be opened, as well as any ADCC timers associated with that instance. The callback function names for each timer frame interrupt must then be registered with the driver (Lines 72 to 73). Following this, DMA mode is enabled (Line 74), and the ADCC clocks and chip selects are configured (Lines 75 to 78).

The timers are then configured, both with the ADCC_TRIG0 input as the trigger source. This is separately connected as a trigger slave to the PWM SYNC pulse trigger master in the SetupTRU() function (Lines 92:97) and as shown graphically in Figure 17. The data enumerations used in these function calls are listed in the Analog Devices Enablement Software package driver documentation.

In Line 81, the EventCFG *struct* defined in Line 62 is passed to the adi_adcc_ConfigEvent driver function, and the adi_adcc_SetEventMask driver function then enables or masks the events as required; in this case, all events are enabled. For maximum ADC throughput it is important to enable the dual bit data interface as per Line 83. This means that 16-bit data can be transferred from the ADC in 8 ADC clock cycles. (Note that if this is not enabled, then N_{CK} in Line 76 and T_{CSCS} in Line 77 have to be set to 16 and 17, respectively.)

Memory is then allocated for the data buffers and they are submitted to the ADCC for filling via the adi_adcc_SubmitBuffer call. The adi_adcc_SubmitBuffer API only works in DMA mode, so DMA mode must be set first before using this API. This function is called again by the main application in Lines 105 to 119 to return the buffer to the ADCC control once data has been extracted from it by the application. Finally, when all configurations have been completed, the instances of the timers and ADCC itself need to be enabled.

Lines 92 to 97 contain the setup of the TRU. This involves opening an instance of the TRU, routing the trigger from the PWM SYNC master to the ADCC slave, and enabling the TRU.

As described previously, the processing of the ADC data at the application level is handled via ADCC timer callbacks following an interrupt on completion of the timer events and associated DMA transfers.

Lines 98 to 126 provide the implementation of the callbacks. The buffered data is extracted from the relevant locations within the buffer and saved to the appropriate global variables. In this example, the updated phase current data is then used immediately within the motor control algorithm which is called from the Timer 1 callback via the algorithm call MotorControl() in Line 117.

Note that the servicing of the ADCC event timer interrupt is the only software call that takes place to access the ADCC data. Synchronization and timing all take place at the hardware level.

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

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

/*Event Times in ACLK Cycles*/
#define SMP_TIME0            950
#define SMP_TIME1            950
#define SMP_TIME2            951
#define SMP_TIME3            952
#define SMP_TIME4            0
#define SMP_TIME5            0

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

#define ADC1_VIN00_CTL          0x0F
#define ADC1_VIN01_CTL          0x1F
#define ADC1_VIN02_CTL          0x2D
#define ADC1_VIN03_CTL          0x3D
#define ADC1_VIN04_CTL          0x4D
#define ADC1_VIN05_CTL          0x5D
#define ADC1_VIN06_CTL          0x6D
#define ADC1_VIN07_CTL          0x7D

/*Mapping the Signals to the Appropriate ADC
Channels*/
#define ES_CTL                 ADC0_VIN00_CTL
#define EC_CTL                 ADC1_VIN00_CTL
#define VDC_CTL                ADC0_VIN02_CTL
#define THS_CTL                ADC0_VIN03_CTL
#define IV_CTL                 ADC0_VIN01_CTL
#define IW_CTL                 ADC1_VIN01_CTL

/*Locations of ADC Signals in Data Buffer Index*/
#define IV_ADC                  0
#define IW_ADC                  1
#define ES_ADC                  0
#define EC_ADC                  1
#define VDC_ADC                 2
#define THS_ADC                 3

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

/*************Function Prototypes****************/
void SetupADC(void);
void SetupTRU(void);
static void AdccTmr0Callback(void *pCBParam,
uint32_t Event, void *pArg);
static void AdccTmr1Callback(void *pCBParam,
uint32_t Event, void *pArg);

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

/*Set Up Event Configuration Table*/
ADI_ADCC_EVENT_CFG  EventCFG[NO_OF_EVENTS] = {
{0,  ES_CTL, ADI_ADCC_ADCIF0, ADI_ADCC_TIMER0,
true,  0, SMP_TIME0},
{1,  EC_CTL, ADI_ADCC_ADCIF1, ADI_ADCC_TIMER0,
true,  2, SMP_TIME1},
{2,  VDC_CTL, ADI_ADCC_ADCIF0,
ADI_ADCC_TIMER0, false,  4, SMP_TIME2 },
{3,  THS_CTL, ADI_ADCC_ADCIF0,
ADI_ADCC_TIMER0, false,  6, SMP_TIME3 },
{4,  IV_CTL, ADI_ADCC_ADCIF0, ADI_ADCC_TIMER1,
true,  8, SMP_TIME4 },
{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*/
result = adi_adcc_OpenDevice(ADCC_DEVICE_NUM,
ADCCMemory, &hADCC);
result = adi_adcc_OpenTimer(hADCC,
ADI_ADCC_TIMER0, ADCCTmr0Memory,
&hADCCTimer0);
result = adi_adcc_OpenTimer(hADCC,
ADI_ADCC_TIMER1, ADCCTmr1Memory,
&hADCCTimer1); /* ADCC Device handle, Timer to
open, Timer memory, Pointer to the timer
handle */
result = adi_adcc_RegisterTmrCallback
(hADCCTimer0, AdccTmr0Callback, hADCCTimer0);
result = adi_adcc_RegisterTmrCallback
(hADCCTimer1, AdccTmr1Callback,
hADCCTimer1);/*Register callback functions*/
result = adi_adcc_EnableDMAMode(hADCC,true);
result = adi_adcc_ConfigADCCClock(hADCC,
ADI_ADCC_ADCIF0, false,1u, 8u );
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*/
result = adi_adcc_ConfigChipSelect(hADCC,
ADI_ADCC_ADCIF0, false, 1u, 0u, 9);
result = adi_adcc_ConfigChipSelect(hADCC,
ADI_ADCC_ADCIF1, false, 1u, 0u, 9);/*For each
interface: ADCC handle, ADC interface, active
low, $T_{CSCK}$, $T_{CKCS}$, $T_{CSCS}$*/
result = adi_adcc_ConfigTimer(hADCCTimer0,
ADI_ADCC_TRIG0, true, false);
result = adi_adcc_ConfigTimer(hADCCTimer1,
ADI_ADCC_TRIG0, true, false); /*For each
timer: Timer handle, Timer trigger source,
falling edge trigger, No trigger output */
result = adi_adcc_ConfigEvent(hADCC,
&EventCFG[0], NO_OF_EVENTS);  /*ADCC handle,
Pointer to the event configuration table,
Number of events in the table */
result = adi_adcc_SetEventMask(hADCC,
EVENT_MASK);                             /*
Handle to the device, Enable all events */
adi_adcc_EnableDualBitDataIF(hADCC, true);
/*Dual bit interface allows highest
throughput*/
memset((void *)SampleBuffer0, 0, NUM_SAMPLES0
* sizeof(short));
memset((void *) SampleBuffer1, 0, NUM_SAMPLES1
* sizeof(short));
result = adi_adcc_SubmitBuffer(hADCCTimer0,
SampleBuffer0, FRAME_INC0, FRAMES_IN_BUFFER);
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 */

result = adi_adcc_EnableTimer(hADCCTimer0,
true);
result = adi_adcc_EnableTimer(hADCCTimer1,
true);
result = adi_adcc_EnableDevice(hADCC, true);
 /*Enable everything*/
}

/*******Function to Configure TRU****************/
void SetupTRU(void){
ADI_TRU_RESULT result;
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*/
result = adi_tru_TriggerRoute (hTru,
TRGS_ADCC0_TRIG0, TRGM_PWM0_SYNC);  /*TRU
device, slave, master*/
result = adi_tru_Enable (hTru, true); /*Enable
TRU*/
}

/***********ADCC Timer Callbacks*****************/
static void AdccTmr0Callback(void *pCBParam,
uint32_t Event, void *pArg){
  switch(Event){
         case ADI_ADCC_EVENT_FRAME_PROCESSED:
           Es_adc= SampleBuffer0[ES_ADC];
           Ec_adc = SampleBuffer0[EC_ADC];
           Vdc_adc = SampleBuffer0[VDC_ADC];
           Ths_adc = SampleBuffer0[THS_ADC];
/*Store all of the data sampled in appropriate
global variables*/
   _adcc_SubmitBuffer(hADCCTimer0,
SampleBuffer0, FRAME_INC0, FRAMES_IN_BUFFER);
/*Return the buffer to the ADCC for use in the
next events*/
           break;
       case ADI_ADCC_EVENT_BUFFER_PROCESSED:
           break;
       default:
          break;
     }

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

     switch(Event){
        case ADI_ADCC_EVENT_FRAME_PROCESSED:
          Iv_adc = SampleBuffer1[IV_ADC];
          Iw_adc = SampleBuffer1[IW_ADC];
          MotorControl();     /*Run the
current control algorithm*/

          adi_adcc_SubmitBuffer(hADCCTimer1,
SampleBuffer1, FRAME_INC1, FRAMES_IN_BUFFER);
          break;

       case ADI_ADCC_EVENT_BUFFER_PROCESSED:
          break;
       default:
          break;
     }
   return;
}
```

EXAMPLE EXPERIMENTAL RESULTS

The current sampling portions of the code described in the Example Code section have been tested in a closed-loop permanent magnet synchronous motor control application circuit. The application circuit operates over a universal ac line input and over a controlled motor current range of −6.8 A to +6.8 A utilizing the current transducer referred to in the current scaling data in Figure 4. Some sample results from the application circuit are shown in this section.

Figure 19 displays the measured motor phase current with a 1500 rpm speed reference and no load on the motor. As expected, the motor current level is very low, and highly discontinuous.

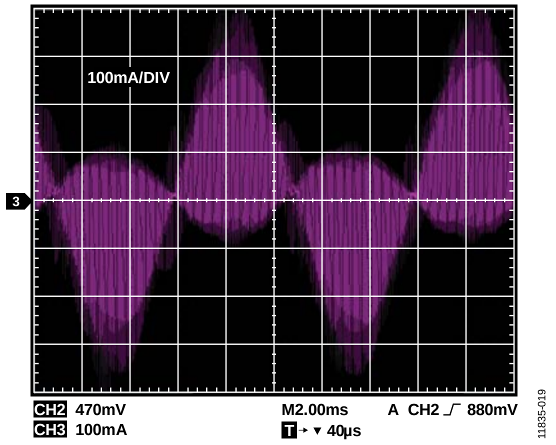

Figure 19. Measured Motor Phase Current

The averaging effect of the correctly synchronized sampling approach is evident in Figure 20, where the smooth sinusoidal averaged shape of the motor phase currents is evident, even at current levels <2% of maximum. This plot, as well as Figure 21 which shows the operation of the control loop in tracking the I_Q reference current, is generated from data streamed from the ADSP-CM408F over RS-232 to a MATLAB® interface.

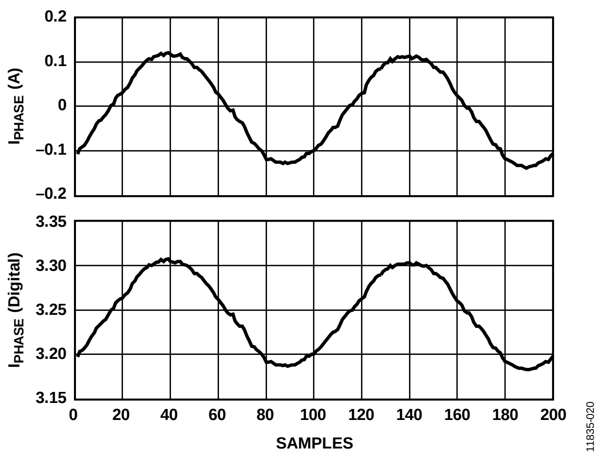

Figure 20. ADC Sampled Motor Phase Current: (Top) Scaled to Real-World Value and (Bottom) Digital Word Output

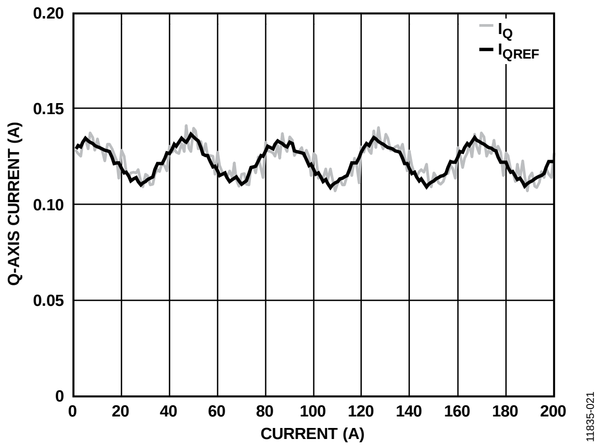

Figure 21. Q-Axis Reference Current and Actual Current

Finally, in Figure 22, the location of the PWM SYNC pulse—and consequent triggering of sampling—is shown at the center of the phase current PWM cycle where the current is equal to the instantaneous average. This is shown at a higher load for ease of illustration.

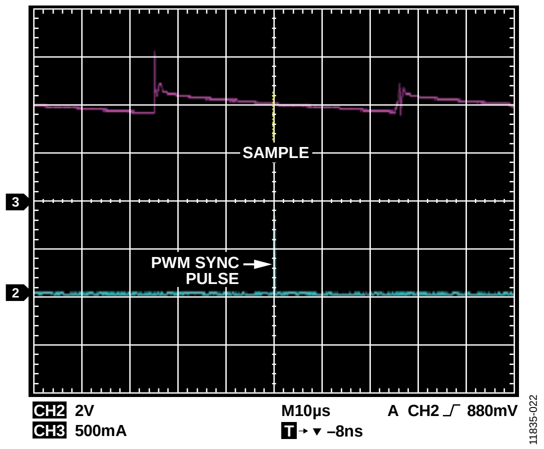

Figure 22. Sampling in Relation to Phase Current

NOTES

NOTES

NOTES

I^{2}C refers to a communications protocol originally developed by Philips Semiconductors (now NXP Semiconductors).

www.analog.com