

## **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 can be found in the [ADSP-CM40x Mixed-Signal Control Processor with ARM Cortex-M4 Hardware Reference Manual](#) available on the [ADSP-CM402F/ADSP-CM403F/ADSP-CM407F/ADSP-CM408F](#) product pages, and on the [ADSP-CM40x Mixed-Signal Control Processor with ARM Cortex-M4 and 16-bit ADCs Development Products](#) product page.

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

Introduction .....	1	ADC Data Access .....	10
Revision History .....	2	ADCC Data Fault Detection.....	10
Current Feedback System Overview.....	3	ADCC Module, Trigger Routing, and Memory Setup.....	11
ADC Module Overview.....	4	Configuration of ADCC Events .....	11
Current Feedback Scaling.....	5	Interrupts and Trigger Routing .....	12
ADC Timing Considerations .....	6	Data Access and Memory Allocation .....	12
ADCC Event Timing.....	6	ADCC Software Support .....	13
ADC Operational Timing .....	7	Example Code.....	13
ADC Pipelining .....	9	Example Experimental Results .....	18

## REVISION HISTORY

### 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

## 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 the Figure 1 setup, the processor is located on the safe, low voltage side of the isolation barrier, with signal isolation typically being inherent to CT0 and CT1, and with digital isolation also existing between the pulse width modulation (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.

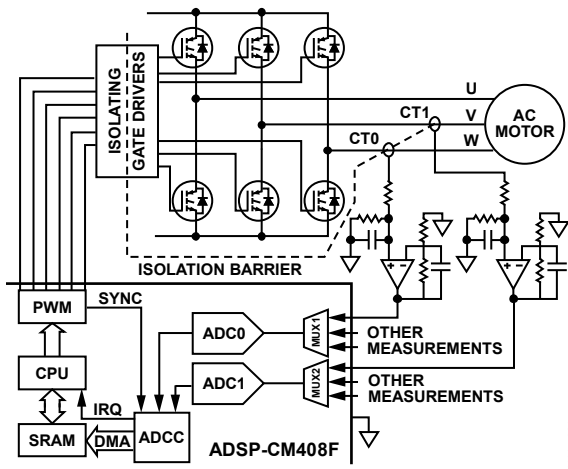


Figure 1. Current Feedback to ADSP-CM408F ADC in Motor Control

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.

In Figure 2, simultaneous U-phase and V-phase sampling is shown occurring at both the zero vector midpoint and the PWM cycle midpoint.

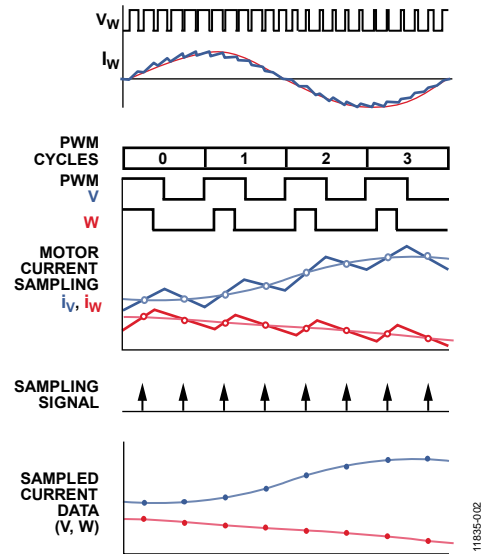


Figure 2. Illustration of Average Current Sampling

When conversion of the data is complete, it can be transferred via direct memory access (DMA) to the controller static random access memory (SRAM), and an interrupt is generated upon 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, isolated gate bipolar transistor (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 has 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 for 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 DMA transfers of sampled data to SRAM.

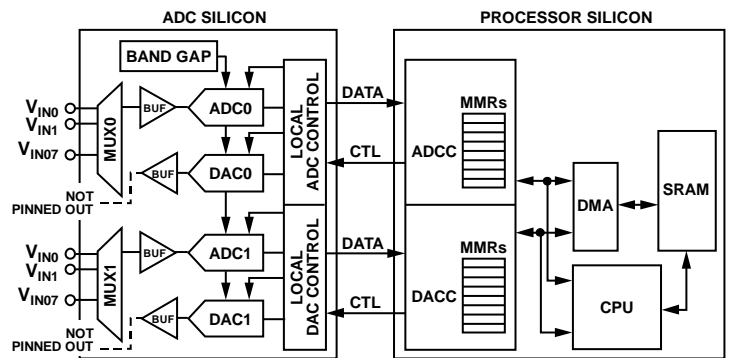


Figure 3. [ADSP-CM408F](#) Analog Subsystem

## CURRENT FEEDBACK SCALING

To correctly use 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 the 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.

$K_{CT}$  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.

Some offset is associated with the output of the ADC; 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  $I_W$ , which is related to the actual phase winding current as

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

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 must 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. The current levels of Figure 4 are converted to scaled quantities (see Table 1) that propagate through the signal measurement system, which is shown in Figure 5.

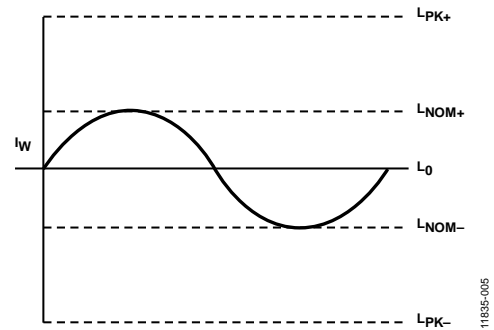


Figure 4. Current Feedback Signal Amplitudes

Table 1. Current Feedback Signal Amplitudes

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 <sub>0</sub>	0	2.5	1.25	0x8000
L-NOM-	-4	+1.25	+0.625	0x4000
L-PK-	-6.8	+0.375	+0.188	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.

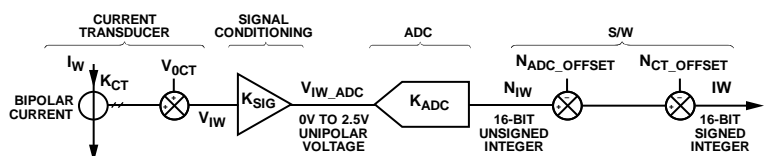


Figure 5. Scaling Relationships in Current Feedback Path

## 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. When 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 (IRQ) is generated and alerts the CPU that a data sample is available.

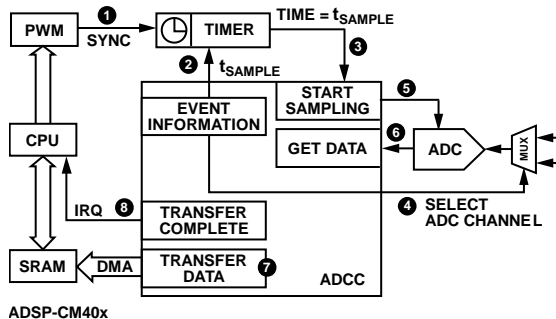


Figure 6. ADCC Operation Sequence

## 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 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 Event x, an event time, TIME<sub>x</sub>, control information shown as CTL<sub>x</sub>, and its resultant data. The event control information, depicted as CTL<sub>x</sub> in Figure 7, contains information for each sample event, such as the ADC interface and channel numbers, the ADC timer 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, CH<sub>x</sub>, 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 the 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.

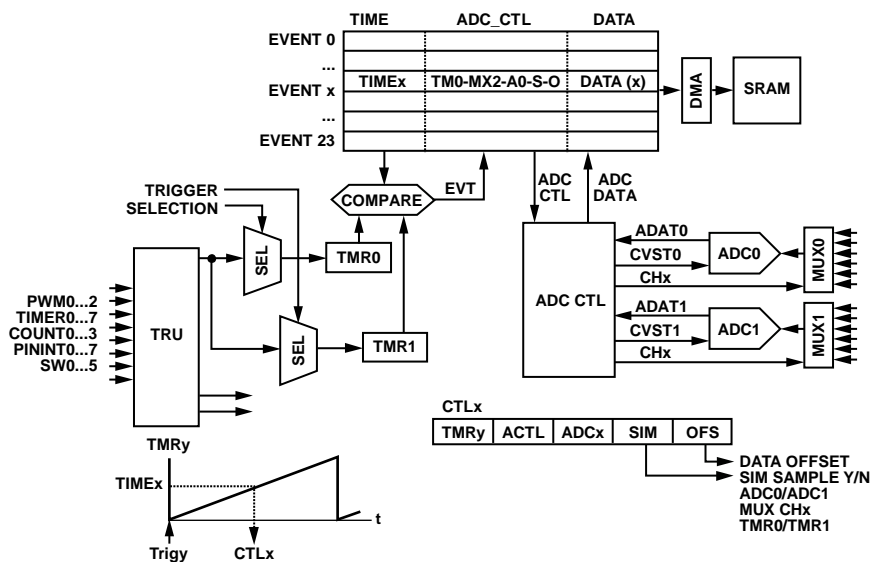


Figure 7. ADCC Module Functional Diagram

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. 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 ( $t_{ACLK}$ ). If Event 2 is the final event associated with Timer 0, the timer stops running after the event is handled to save power.

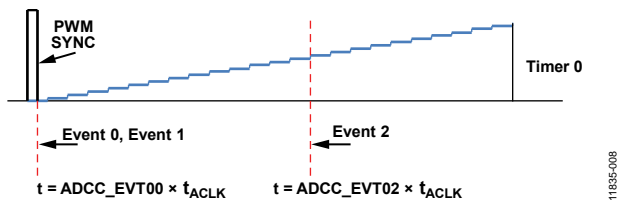


Figure 8. Event Timing

### 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 ADCC.

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 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 ( $t_{CSCS}$ ), and the minimum delays between CS edges and ACLK edges ( $t_{CSCK}$  and  $t_{CKCS}$ ). Consequently, the ADC conversion cycle time,  $t_{CONV\_ADC}$ , for a single pair of simultaneously sampled signals is given by

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

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

The ADCC clock is internally generated from the processor system clock ( $f_{SYSCLK}$ ) by means of the divisor ACKDIV (in the 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.

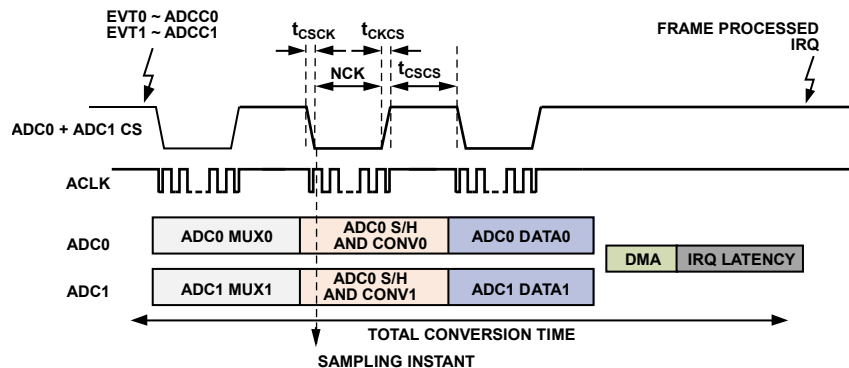


Figure 9. Conversion Timing of a Simultaneous Sampling Single Event

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 at least 380 ns must be allowed for the ADC sampling and conversion cycle ( $t_{CONV\_ADC}/3$ ). The resulting 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).

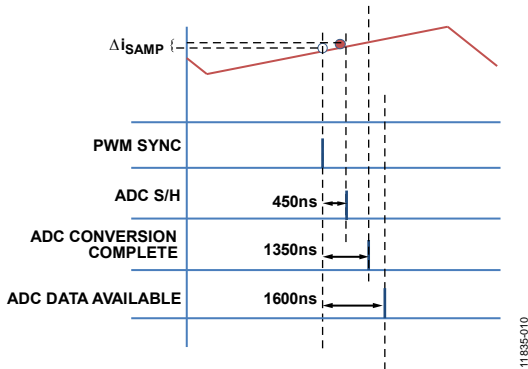


Figure 10. Sample Delay Times

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  $\Delta i_{SAMP}$  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  $<0.2^\circ$  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.

**Adjustment of Sampling Instant**

It may be important to further enhance the precision of the motor current sampling instant and to remove the 450 ns offset between desired and actual sampling instant. Use cases such as low inductance servo motors, or situations where higher switching frequencies are being utilized would especially benefit from this enhanced precision. One option to cancel this small time offset is to use a general purpose (GP) timer to create a trigger at a point in time one ADCC chip select pulse width before the PWM sync pulse. This can be achieved by triggering the GP timer from the previous PWM sync pulse, as illustrated in Figure 11.

With this approach, care must be taken when scheduling any sampling events towards the end of the PWM cycle. All sampling events must be completed one chip select pulse width before the beginning of the next cycle (EVT0 marker in Figure 11).

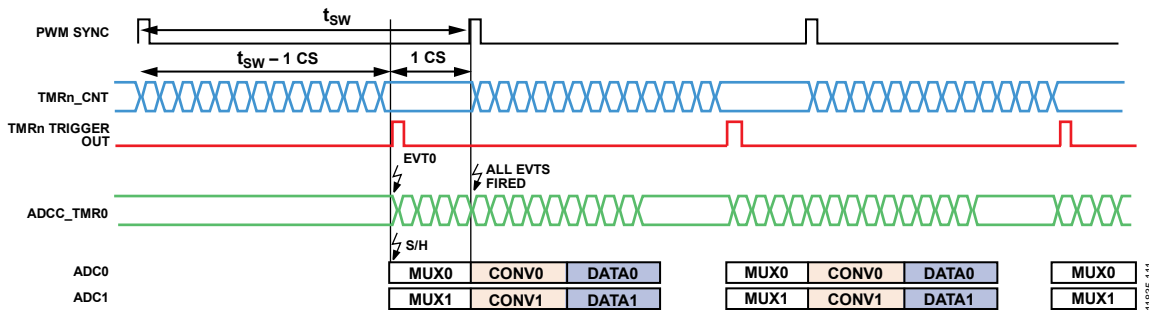


Figure 11. Implementation of Sampling Instant Adjustment

Table 2. Timing Settings for a Typical ADC Setup

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 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, first in, first out (FIFO) buffer, one of which is available for each ADC interface. When the control word is 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 occurs. Likewise, a second pending event has its control word phase initiated upon 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 pipelining is illustrated in Figure 13 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.

Figure 7 shows that during one of the CS assertions, the ADCC handles 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 time spacing corresponds to 450 ns for the settings in Table 2 and can 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 approach ensures that new ADC samples are available at the earliest opportunity within the PWM cycle. Implementation of the pipelining shown in Figure 13 requires that all of the event times are close to zero, that is, immediately after the PWM sync pulse.

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 12 for different simultaneously sampled pair numbers with the timing settings shown in Table 2.

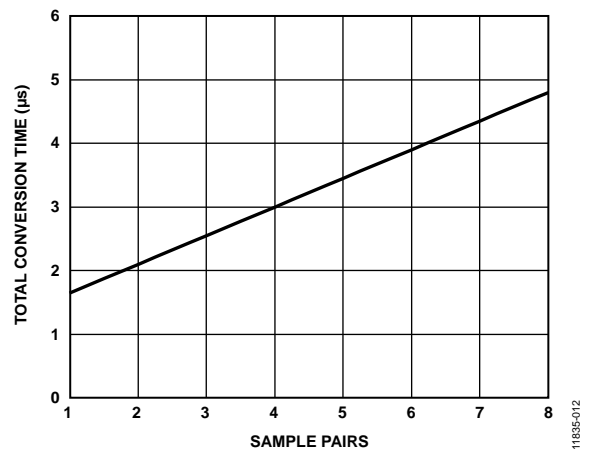


Figure 12. Total Conversion Time for Different Sample Pair Numbers

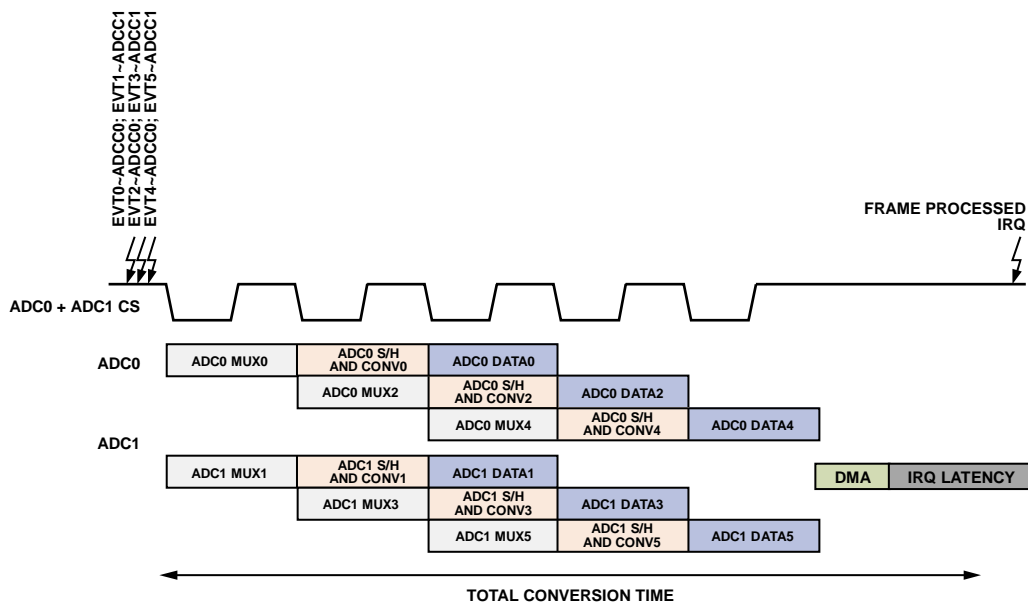


Figure 13. Pipelining of Events Within the ADC

## ADC DATA ACCESS

The examples shown thus far have all assumed that the ADC data is accessed in the memory via an automatic DMA transfer. Data access directly from core reads of the ADCC memory mapped registers (MMRs) is also possible, as shown in Figure 14. Note that ACK in Figure 14 represents an acknowledge signal, not the analog clock.

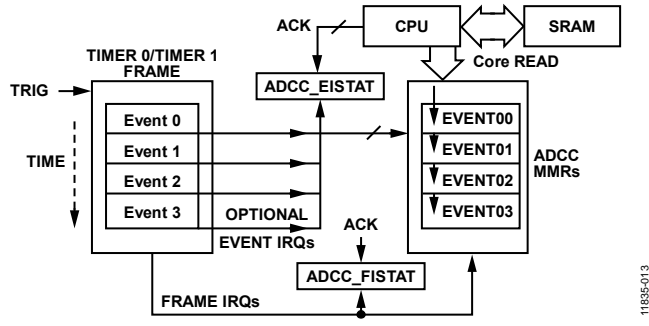


Figure 14. ADC Data Access in Core Mode

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 shown in Figure 15. 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.

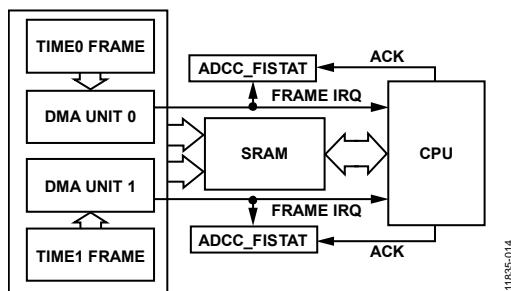


Figure 15. ADC Data Access in DMA Mode

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 is 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 is not 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 can 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 is used, 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 that the processor can execute the control algorithm and update the PWM modulator registers.

Figure 17 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 and cosine signals, heat sink temperature, and dc bus voltage are provided as examples of additional monitoring inputs.

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.

### CONFIGURATION OF ADCC EVENTS

Configuration of the ADCC events for the example shown in Figure 17 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; one possibility is shown in Figure 16 and listed in Table 3. This example utilizes both timers for illustration purposes only.

For this specific example, the events can 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.

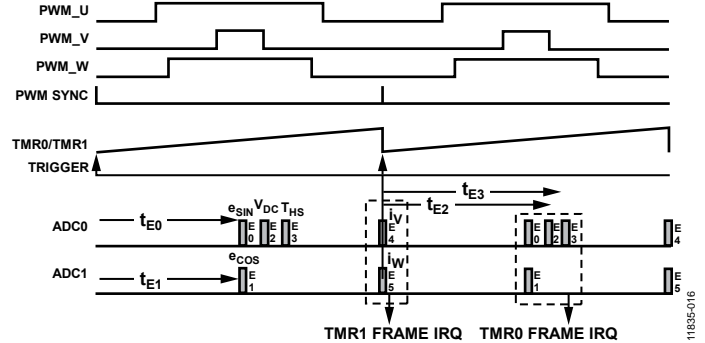


Figure 16. Typical ADCC Use in a Motor Control Application

Table 3. Event Configurations for Example Application

Event	Timer	ADC I/F	ADC Ch	Time	Simultaneous Sample
E0 ( $e_{SIN}$ )	TMR0	0	0	$t_{E0}$	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

Phase currents  $i_V$  and  $i_W$  are simultaneously sampled immediately after the PWM sync pulse trigger has occurred, and these phase currents 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 and 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.

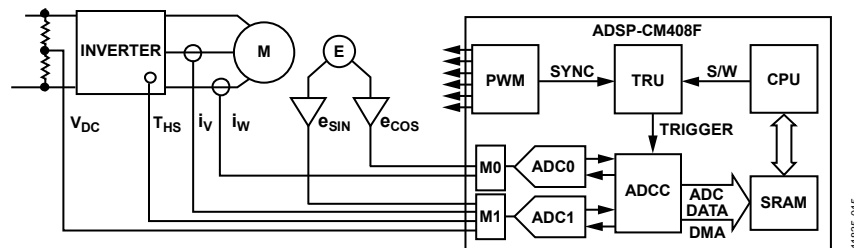


Figure 17. System Interconnections in Typical Motor Control Application

## INTERRUPTS AND TRIGGER ROUTING

In the example in Figure 17, all events in time are referenced to the PWM cycle; therefore, 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.

The routing of the appropriate triggers is shown conceptually in Figure 18 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 setting the appropriate value for the TRIGSEL bits in the ADC\_CTL register.

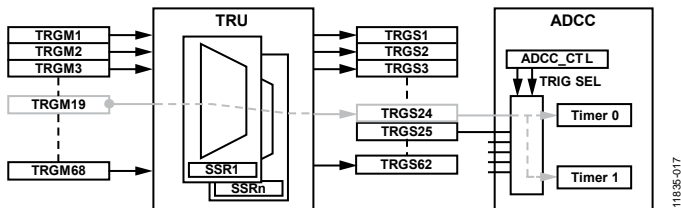


Figure 18. Trigger Routing from PWM Sync to ADCC Timers

This trigger routing arrangement 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, and counter events. This arrangement 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.

### Trigger Routing for Enhanced Precision Sample Timing

Removal of the chip select pulse width lag from the current sample timing, as described previously, requires slightly different arrangement of the trigger routing. In this case, the ADCC timers are triggered from a GP timer trigger, which is itself triggered from PWM sync. This sequence can be seen in Figure 11.

## DATA ACCESS AND MEMORY ALLOCATION

As illustrated in Figure 14 and Figure 15, the ADC data can be accessed either via core MMR reads or by making it available in SRAM by DMA transfer. 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 19 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, the ADCC\_FRINC register contains the offset for the pointer to the base of the next frame. In linear buffering mode, which is activated by writing zero to the ADCC\_CBSIZ register, additional frames are stored in the memory in a continuously increasing linear manner, spaced by the frame increment value. If a nonzero value, M, is written to the ADCC\_CBSIZ, circular buffering is activated, and M frames are written to the memory before the frame base pointer returns to the ADCC\_BPTR value and begins to overwrite the existing frames.

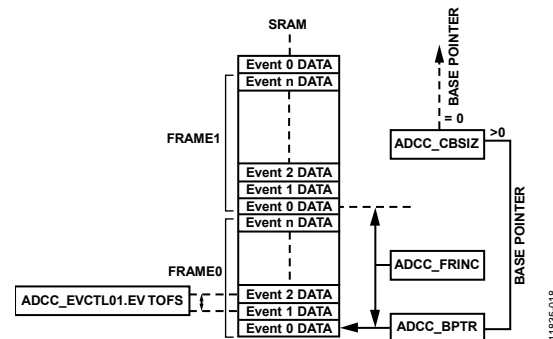


Figure 19. Memory Configuration for ADC DMA Transfers

In the motor control application example in Figure 17, 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 the samples in a linear manner because memory is very quickly overloaded. In such an application, it is better to enable circular buffering with M limited to 1 or some small value, or to set the ADCC\_FRINC value to 0 and overwrite the frame every PWM cycle. The driver application programming interfaces (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.

### 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 17. 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.

Line 1 through Line 10 define the frame and associated data buffer sizes for each timer. The factor 2 is included in allocating the sample buffer lengths as a safety measure for debugging purposes. Because ADC sample transfer to memory is entirely hardware triggered (including DMA), inserting a software breakpoint at Line 122, before a new buffer is submitted to the driver and ADC buffer pointer are reset, can cause memory to be overwritten. Allowing an additional buffer for headroom prevents this debugging related issue from occurring. The number of frames in the buffer is defined as 1, which 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.

Line 11 through Line 16 define the sample times for each event in ACLK cycle numbers as shown in 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.

Line 17 to Line 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.

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

Line 60 to Line 91 contain the main ADCC configuration function SetupADC(). The first step is to set up the event configuration table, a struct that 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 (Line 72 to Line 73). Following this, DMA mode is enabled (Line 74), and the ADCC clocks and chip selects are configured (Line 75 to Line 78).

The timers are then configured, both with the ADCC\_TRIG0 input as the trigger source. The ADCC\_TRIG0 trigger is separately connected as a trigger slave to the PWM sync pulse trigger master in the SetupTRU() function (Line 92 to Line 97) and as shown graphically in Figure 18. 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, meaning that 16-bit data can be transferred from the ADC in eight ADC clock cycles. (Note that if the dual-bit interface is not enabled, N<sub>CK</sub> in Line 76 and t<sub>CSCS</sub> in Line 77 and Line 78 must be set to 16 and 17, respectively.) Memory is then allocated for the data buffers, and the data buffers are submitted to the ADCC for filling via the adi\_adcc\_SubmitBuffer call. The adi\_adcc\_SubmitBuffer API only works in DMA mode; therefore, DMA mode must be set first before using this API. This driver function is called again by the main application in Line 105 to return the buffer to the ADCC control once data is extracted from it by the application. Finally, when all configurations are completed, the instances of the timers and ADCC itself need to be enabled.

Line 92 to Line 97 contain the setup of the TRU, which 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.

Line 98 to Line 127 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.

Line 128 to Line 136 provide additional code snippets that can be inserted into the TRU and ADCC setup functions to enable the enhanced precision sample timing functionality depicted in Figure 11. In Line 128 to Line 129, the hardware trigger routing

path from PWM sync to GP timer TMR7 to ADCC Timer 0 trigger is set up. Line 130 to Line 136 provide sample code that can be inserted within the ADC setup function to correctly configure and enable the GP timer TMR7 to provide the correct delay.

In all cases, the SetupTRU function call must occur prior to calling the SetupADC function.

```

/*****
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_INCO
2*NUM_SAMPLES0*sizeof(short)
7. #define FRAME_INCL
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, ADCC_Tmr0Memory,
    &hADCCTimer0);
71. result = adi_adcc_OpenTimer(hADCC,
    ADI_ADCC_TIMER1, ADCC_Tmr1Memory,
    &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_ConfigADCClock(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, Tcsc, 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*/

```

**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, using the current transducer referred to in the current scaling data in Figure 4. Sample results from the application circuit are shown in Figure 20 to Figure 23.

Figure 20 displays the measured motor phase current with a 1500 rpm speed reference and no load on the motor. The motor current level is very low and highly discontinuous.

The averaging effect of the correctly synchronized sampling approach is shown in Figure 21, where the smooth sinusoidal averaged shape of the motor phase currents is evident, even at current levels <2% of maximum. Figure 21, as well as Figure 22, 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 an RS-232 to a MATLAB® interface.

In Figure 23, 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 plot is shown at a higher load for ease of illustration.

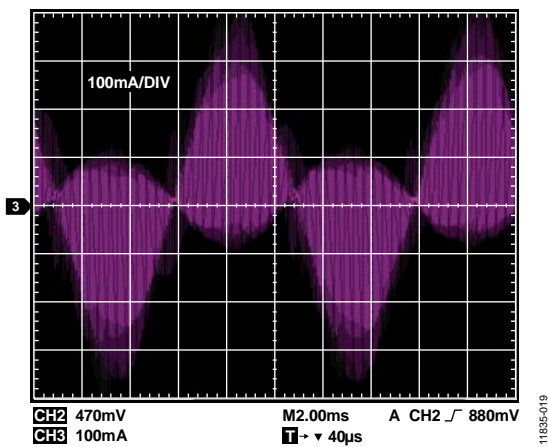


Figure 20. Measured Motor Phase Current

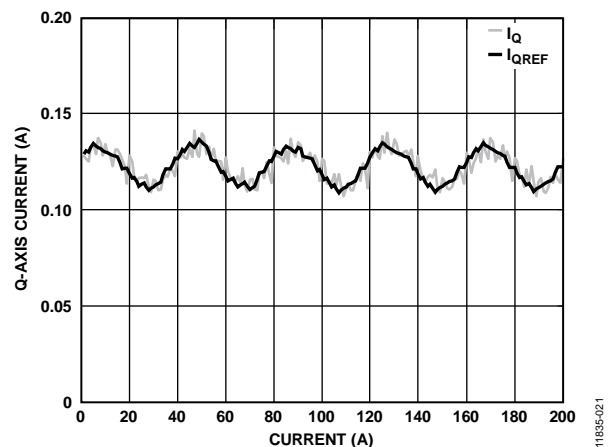


Figure 22. Q-Axis Reference Current and Actual Current

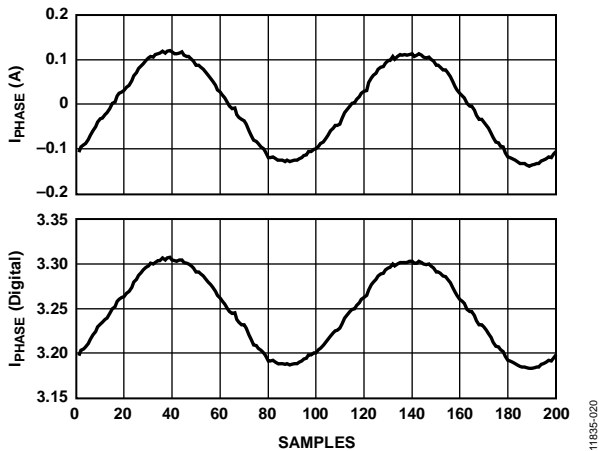


Figure 21. ADC Sampled Motor Phase Current: (Top) Scaled to Real-World Value and (Bottom) Digital Word Output

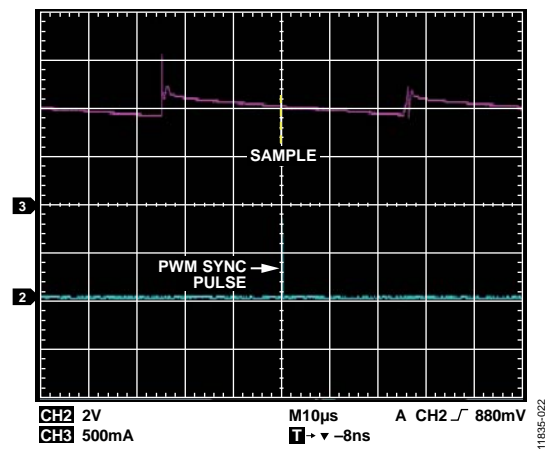


Figure 23. Sampling in Relation to Phase Current