Application Report
SPRACS0–May 2020

Software Examples to Showcase Unique Capabilities of TI's C2000™ CLA

Himanshu Chaudhary and Aravindhan Karuppiah

ABSTRACT

Enabling extremely high performance computation and efficient processing is critical for solving today's complex real-time control problems. Real-time control systems are closed-loop control systems where one has a tight time window to gather data, process that data, and update the system in order to meet the performance objectives. TI's Control Law Accelerator (CLA) is designed to execute real-time control algorithms in parallel with the C28x CPU, effectively doubling the computational performance of C2000 devices. This application report discusses some of the unique features of CLA and demonstrates them using simple software examples. These stand-alone examples are available as part of C2000Ware and can be quickly used to explore and evaluate the capabilities of CLA.

Contents

List of Figures

List of Tables

Trademarks

1 Introduction

The CLA is a fully-programmable independent 32-bit floating-point CPU that is designed for optimal math intensive computations to offer a significant boost to the performance of control algorithms. Unlike the standard traditional processor which executes instructions and services interrupts, the CLA instead is a task-driven machine and can support up to 8 user-defined tasks. The CLA in addition to providing computational capability provides an unique combination of minimal latency and ease of access to the key control peripherals. This makes the CLA ideal for implementing fast control loops, thus freeing up bandwidth on C28x to run additional control loops and perform other diagnostic and communication related tasks. The subsequent sections of this application report discusses these unique capabilities of CLA in detail and also demonstrates them through simple software examples which are provided as part of C2000Ware package [2]. For more details on CLA architecture and instructions set, see [1], [3].

The examples discussed in this document can be found in C2000Ware v3.01.00.00 or latest, located within the following directories after installation:

- C:\ti\c2000\C2000Ware_<version_number>\driverlib\f28004x\examples\cla
- C:\ti\c2000\C2000Ware_<version_number>\libraries\math\CLAmath\c28\examples
- C:\ti\c2000\ C2000Ware_<version_number>\libraries\control\DCL\c28\examples

The discussed example projects are:

- cla_ex4_pwm_control
- cla_ex5_adc_just_in_time
- cla_ex6_cpu_offloading
- cla_ex7_shared_resource_handling

2 Direct Access of CLA to Key Peripherals

Most of the real-time control algorithms can be split into three main tasks: excite the system, sample the system and control the system. Exciting the system would involve updating the PWM registers, sampling the system involves accessing the ADC result registers while controlling the system involves control loop math computations. CLA being an independent math processor, also has the ability to access registers of all key peripherals used for control applications like EPWM, ADC, ECAP, EQEP, CMPSS, and so forth directly. This allows CLA to perform sampling and actuation along with computation of control logic and is capable of executing the entire control task independently without any C28x involvement.

The example “cla_ex4_pwm_control” showcases how to control the PWM signal output directly through CLA. The block diagram of this example is shown in Figure 1. In this example, EPWM1 is configured to generate complementary signals on both of its channels at a fixed frequency of 100 KHz while EPWM4 is configured to trigger a periodic CLA control task at a frequency of 10 KHz. The CLA Task 1 implements a very simple logic to vary the duty of the EPWM1 outputs by increasing it by 0.1 for every iteration while maintaining it in the range of 0.1-0.9. The code sequence below illustrates how the existing C28x driverlib APIs (available as part of C2000Ware) can be used as it is within the CLA task to update the EPWM registers avoiding any additional software development effort with respect to CLA. The CLA task can access key registers of other shared peripherals as well in a similar fashion. Note that the CLA global variables cannot be initialized at the start of .cla file thus this example also illustrates a systematic way of initializing all the CLA global variables inside a dedicated CLA task (CLA task 8), which is triggered by C28x software at the time of initialization.

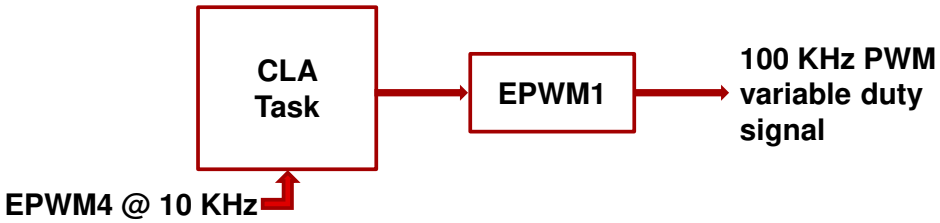

Figure 1. Direct PWM Control Using CLA

```
__attribute__((interrupt)) void Cla1Task1 ( void )
{
    //
    // Uncomment this to debug the CLA while connected to the debugger
    //
    __mdebugstop();

    //
    // Write to the COMPA register to realize a particular duty value
    //
    EPWM_setCounterCompareValue(EPWM1_BASE, EPWM_COUNTER_COMPARE_A,
                                (uint16_t)(duty * EPWM1_PERIOD + 0.5f));

    //
    // Update duty value and use the limiter
    //
    duty += 0.1f;
    duty = (duty > 0.9f) ? 0.1f : duty;

    //
    // Clear EPWM4 interrupt flag so that next interrupt can come in
    //
    EPWM_clearEventTriggerInterruptFlag(EPWM4_BASE);
}
```

3 Low interrupt Latency of CLA

In any real-time control application, the sample to output delay, defined as the time that elapses between sensing, processing and actuation, is an important system consideration. The low-latency architecture of CLA reduces this sample to output time while increasing the overall system throughput. This is made possible because CLA is task oriented instead of interrupt driven machine and does not use interrupts to synchronize with hardware. Instead, it supports up to eight independent tasks, which are each mapped to hardware events such as a timer or data availability on an ADC, and so forth. A task initiated on the CLA runs to completion without any interruption or nesting involved, hence eliminating the need for any context-switching overhead typically involved in traditional interrupt-based processors. Thus, there is little to no delay involved in processing the data by CLA, which ultimately reduces the sample to output delay and enables faster system response. Figure 2 illustrates the differences between a task driven machine (TDM) and an interrupt driven machine (IDM).

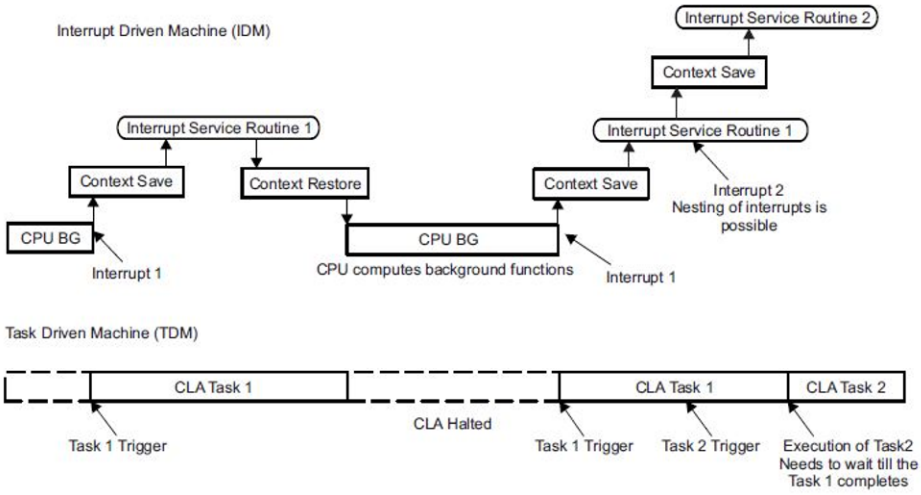

Figure 2. Interrupt vs Task Driven Machine

The low interrupt response of CLA can be leveraged in combination with the early-interrupt feature of TI's internal ADC to further reduce the sample to output delay. The ADC can be configured to generate an early interrupt pulse at the end of sampling before the conversion completes. This early-interrupt pulse from the ADC can be used to trigger a CLA task that would allow the CLA to read the result as soon as the conversion result is available in the ADC result register. This combination of just-in-time sampling along with the low interrupt response of the CLA enable faster system response and higher frequency control loops. The available time before the conversion can be effectively utilized for any necessary pre-processing steps within the CLA task as illustrated in Figure 3. The exact instruction at which the read request should be placed to achieve just-in-time read can be calculated based on the CLA pipeline activity for N-cycle ADC conversion. As shown in Figure 4, The N-2 instruction will arrive in the R2 phase just in time to read the result register. For the standard 12-bit ADC configuration and clock divider as 4, N is 42. To find out the correct value of N based on the configuration of ADC, see the device-specific data sheet [4].

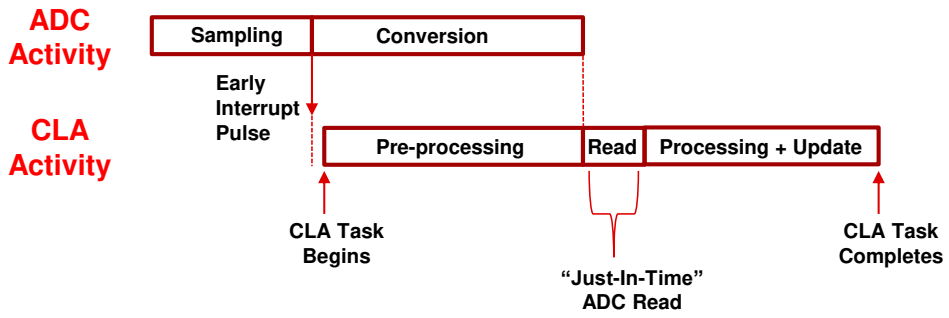

Figure 3. Early Interrupt From ADC to Trigger CLA Task

ADC Activity	CLA Activity	F1	F2	D1	D2	R1	R2	E	W
Sample									
Sample									
...									
Sample									
$Conversion_{(Cycle\ 1)}$	Interrupt Received								
$Conversion_{(Cycle\ 2)}$	Task Startup								
$Conversion_{(Cycle\ 3)}$	Task Startup								
$Conversion_{(Cycle\ 4)}$	$I_{(Cycle\ 4)}$	$I_{(Cycle\ 4)}$							
$Conversion_{(Cycle\ 5)}$	$I_{(Cycle\ 5)}$	$I_{(Cycle\ 5)}$	$I_{(Cycle\ 4)}$						
$Conversion_{(...)}$	...	...	...	...	...	...	...		
$Conversion_{(Cycle\ N-6)}$	$I_{(Cycle\ N-6)}$	$I_{(Cycle\ N-6)}$	$I_{(Cycle\ N-7)}$	$I_{(Cycle\ N-8)}$	$I_{(Cycle\ N-9)}$	$I_{(Cycle\ N-10)}$	$I_{(Cycle\ N-11)}$		
$Conversion_{(Cycle\ N-5)}$	$I_{(Cycle\ N-5)}$	$I_{(Cycle\ N-5)}$	$I_{(Cycle\ N-6)}$	$I_{(Cycle\ N-7)}$	$I_{(Cycle\ N-8)}$	$I_{(Cycle\ N-9)}$	$I_{(Cycle\ N-10)}$		
$Conversion_{(Cycle\ N-4)}$	$I_{(Cycle\ N-4)}$	$I_{(Cycle\ N-4)}$	$I_{(Cycle\ N-5)}$	$I_{(Cycle\ N-6)}$	$I_{(Cycle\ N-7)}$	$I_{(Cycle\ N-8)}$	$I_{(Cycle\ N-9)}$		
$Conversion_{(Cycle\ N-3)}$	$I_{(Cycle\ N-3)}$	$I_{(Cycle\ N-3)}$	$I_{(Cycle\ N-4)}$	$I_{(Cycle\ N-5)}$	$I_{(Cycle\ N-6)}$	$I_{(Cycle\ N-7)}$	$I_{(Cycle\ N-8)}$		
$Conversion_{(Cycle\ N-2)}$	**Read RESULT**	**Read RESULT**	$I_{(Cycle\ N-3)}$	$I_{(Cycle\ N-4)}$	$I_{(Cycle\ N-5)}$	$I_{(Cycle\ N-6)}$	$I_{(Cycle\ N-7)}$		
$Conversion_{(Cycle\ N-1)}$			**Read RESULT**	$I_{(Cycle\ N-3)}$	$I_{(Cycle\ N-4)}$	$I_{(Cycle\ N-5)}$	$I_{(Cycle\ N-6)}$		
$Conversion_{(Cycle\ N-0)}$				**Read RESULT**	$I_{(Cycle\ N-3)}$	$I_{(Cycle\ N-4)}$	$I_{(Cycle\ N-5)}$		
Conversion Complete					**Read RESULT**	$I_{(Cycle\ N-3)}$	$I_{(Cycle\ N-4)}$		
RESULT Latched						**Read RESULT**	$I_{(Cycle\ N-3)}$		
RESULT Available							**Read RESULT**		

Figure 4. CLA Pipeline Activity for Early Interrupt Pulse

The example “cla_ex5_adc_just_in_time” utilizes the above concept to read the ADC data “just-in-time” even at very high sampling frequencies. As depicted in Figure 5, EPWM1 is configured to generate a PWM output signal of frequency 1 MHz, which is also used to trigger the ADC sampling at each cycle. The example also utilizes the newly added feature in TI’s Type 5 ADCs, which allows delaying the early interrupt pulse by few cycles as per the programmed OFFSET value. Thus ADCA is configured to sample the input on Channel 0 and to generate the early interrupt at the end of S/H + offset cycles. This interrupt is used to trigger the CLA control task. The CLA task implements the control logic to update the duty of the PWM output based on the read ADC value. The early interrupt feature and low interrupt latency of CLA allows the application to do any necessary pre-work so that the application can act on the ADC results immediately when they become available and still complete updating the PWM output before the next interrupts arrives. Thus, all the three steps (sampling, processing and actuation) are completed within a 1 MHz cycle. As shown in the below code snippet of the CLA task, 3-point moving average filter is used to simulate the processing sequence for illustration purposes and few steps of the filtering sequence that are denoted as the pre-processing code are implemented before reading the ADC result to make use of the time available before conversion.

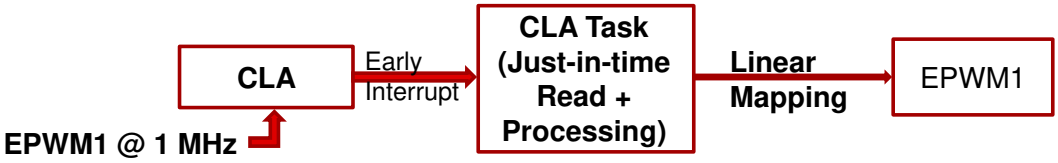

Figure 5. “Just-in-time” ADC Read Example Showcase

```
//
// Pre-processing for implementing moving average filter, takes 13 cycles
// This is just to illustrate how cycles can be utilized to do some pre-
// processing before ADC result latches. Based on the cycles taken by
// pre-processing code, ADC interrupt offset need to be programmed
//
data_read_total = data_read + data_read_prev;
data_read_prev2 = data_read_prev;
data_read_prev = data_read;

//
// Reading ADC just-in-time
//
data_read = HWREGH(ADCARESULT_BASE + ADC_RESULTx_OFFSET_BASE + ADC_SOC_NUMBER0);

//
// "data_read_total" stores the cumulative sum of current and last 2 data elements
//
data_read_total += data_read;

//
// Taking average of 3 elements, normalizing for 12-bit and mapping to output duty
// linearly in the range  0.1-0.9
// duty = 0.1 + (0.9-0.1) * ((data_read_total / 3) / 2^12 )
//
duty = 0.1f + (data_read_total / (15360.0f));

//
// Writing to the COMPA register for realizing computed duty value
//
HWREGH(EPWM1_BASE + EPWM_O_CMPA + 0x1U) = (uint16_t)(duty * EPWM1_PERIOD + 0.5f);
```

The early interrupt OFFSET value of ADC need to be adjusted based on the cycles consumed by the pre-processing in order to read the ADC data “just-in-time”. In this example, the OFFSET value of 20 is used based on the calculation shown in example header. The programming sequence for this configuration of ADC is shown below. The actual use-case may involve different pre-processing steps, hence the interrupt OFFSET value need to programmed accordingly.

```
//
// Set pulse positions to early
//
ADC_setInterruptPulseMode(ADCA_BASE, ADC_PULSE_END_OF_ACQ_WIN);

//
// Set interrupt offset delay as 20 cycles based on the calculation
// shown in example header
//
ADC_setInterruptCycleOffset(ADCA_BASE, 20);
```

4 Powerful Math Computation Capability of CLA

CLA also offers powerful 32-bit floating point processing capability to C2000 devices and provides a significant boost to the performance of typical math functions that are commonly used in control algorithms. The powerful CLA instruction set supports floating point multiplication with parallel add or subtract operations in a single cycle and also supports computation of inverse square root in a single cycle too. For the ease of software development with CLA, a wide collection of commonly used floating-point math functions (a few of them are listed in Table 1) are packaged into a single library called as CLA Math, which is available as part of C2000Ware. This source code library includes several C callable assembly math functions optimally written for CLA architecture.

In addition to the basic math routines, TI also provides Digital Control library (DCL available as part of C2000Ware) that includes optimal implementation of standard control routines on CLA CPU, few of them are listed in Table 1. These C callable assembly control routines can be called within a CLA application task to realize digital controller on CLA CPU. Along with the library source code, examples are provided to show the user how to integrate the library into their projects and use any of the math or control routines. These examples can be found in the example directories indicated in the introduction section that can be used to explore and evaluate the compute capability of CLA.

Table 1. CLA Math and Control Library Routines

Library	Routine	Description	Cycles
CLA Math	CLAcos	Calculates cosine on CLA	28
	CLAsin	Calculates sine on CLA	28
	CLAacos	Calculates arc-cos on CLA	24
	CLAasin	Calculates arc-sine on CLA	22
	CLAatan	Calculates arc-tan on CLA	41
	CLAlog10	Calculates Log (base10) on CLA	29
	CLAexp	Calculates exponential on CLA	41
	CLAdiv	Calculates floating-point division on CLA	13
	CLAisqrt	Calculates inverse square root on CLA	14
	CLAsqrt	Calculates square root on CLA	16
DCL	DCL_runPID_L1	Runs Ideal Form PID controller on CLA	53
	DCL_runPID_L2	Runs Parallel Form PID controller on CLA	45
	DCL_runPI_L1	Runs Ideal Form PI controller on CLA	34
	DCL_runDF13_L1	Runs the DF13 Full Compensator on CLA	61
	DCL_runDF13_L2	Runs the DF13 Immediate Compensator on CLA	20
	DCL_runDF13_L3	Runs the DF13 Partial Compensator on CLA	58

5 Offloading Fast Control Loop to CLA

Various real-time control applications involve implementation of multiple control loops on a single device. However integration of multiple control systems on a single controller remains challenging from a processor bandwidth point of view while keeping system costs down. The CLA is a fully parallel processor to the main C28x core that brings concurrent control-loop execution to the C28x family. The CLA has its own program and data bus, and executes independently of the main core on the MCU. As described in Section 2 and Section 3, CLA provides unique combination of minimal latency and ease of access to the key control peripherals, which enables CLA to offload the fast control algorithm task entirely from C28x. Offloading the control task to CLA also offers additional benefits such as reduced jitter in execution and deterministic operation of control loops. This is made possible because the CLA is task oriented instead of an interrupt service driven machine and the tasks on CLA cannot be interrupted guaranteeing the deterministic nature of control loops. In a pipelined CPU, the ISRs can be delayed by an “n” number of cycles if the CPU is executing branch type statements when the ISR is received. However this is not a problem with CLA CPU as it waits in an idle state till the periodic task triggers to begin any execution due to its task-driven nature. Therefore, offloading fast control task to CLA and running remaining tasks on C28x helps to improve the overall system performance with reduced jitter in execution.

The example “cla_ex6_cpu_offloading” illustrates how to optimally offload a control loop from C28x to CLA when multiple control tasks and background tasks are involved which require more than single CPU (C28x) bandwidth. Figure 6 shows that two control loops are simulated in this example. The faster one (loop1) runs at 200 KHz while the slower one (loop2) runs at 20 KHz. Both the loops make use of PI controller to control the duty of single PWM output with different weightage, the faster one contribution being 80% while the slower one contributes 20% to the PWM output. The inputs for both the loops are sampled using ADCA and ADCB with multiple SOCs for each to filter out any noise in the inputs. There is also a background task continuously running in the main loop that disables or enables the entire system including the PWM output and the control loops based on the user configured switch "system_OFF". Note that the CCS debugger clock cannot be used for profiling CLA routines, hence GPIO based profiling technique is employed in this example to profile the both tasks. GPIO2 and GPIO 3 have been used for this purpose.

Figure 7 depicts the flow diagram of both the control tasks when everything runs on C28x without the use of CLA. In this case, the total CPU (C28x) utilization exceeds the schedulable Utilization bound (UB) and, hence the system is schedulable in this scenario. This can also be further substantiated by observing the profiling waveforms shown in Figure 8. Note that there is no toggling observed on GPIO3, which clearly suggests that the lower priority Loop 2 task never gets chance to complete and neither the background task.

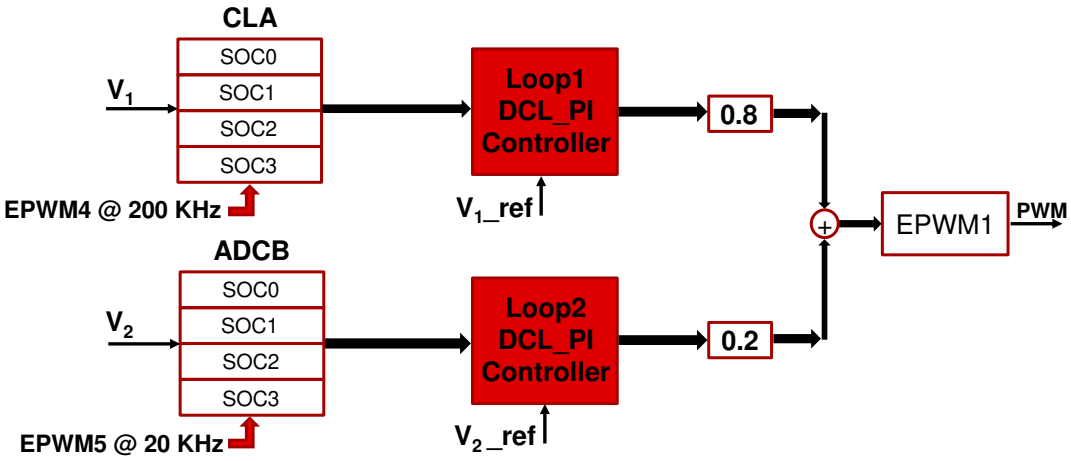

Figure 6. Dual Control Loop Example Showcase

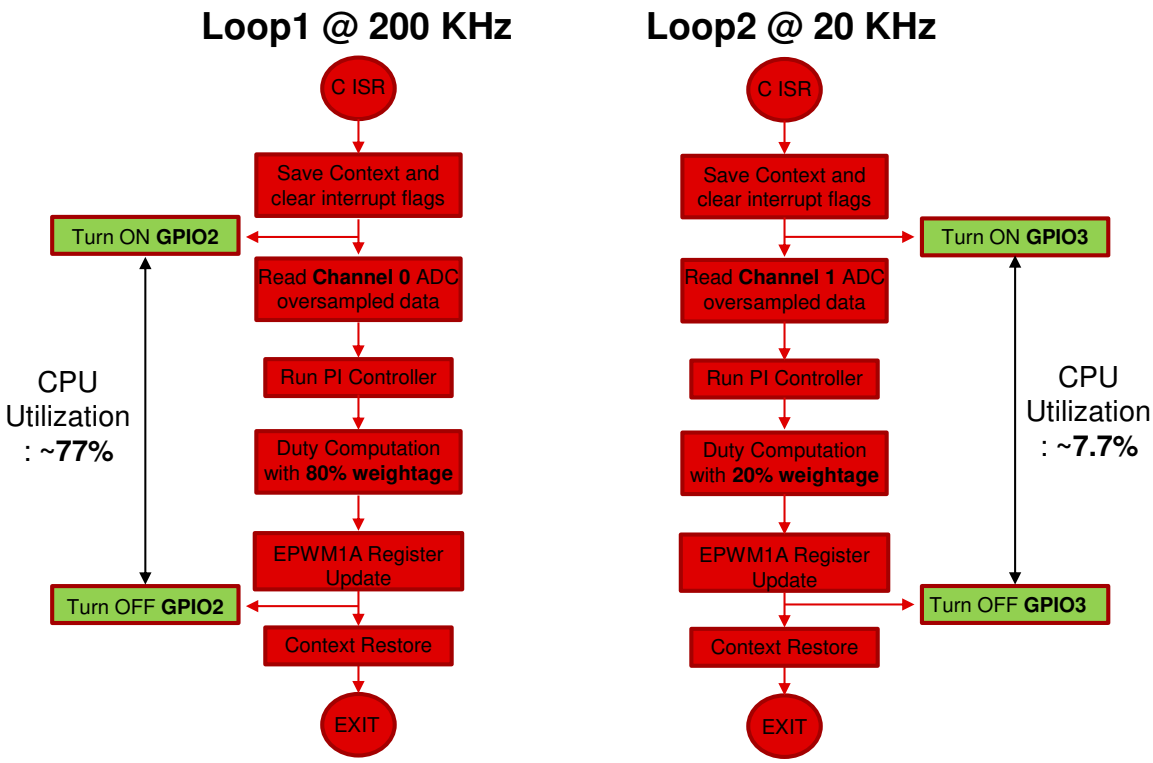

Figure 7. Flow Diagram With Both Tasks Running on C28x

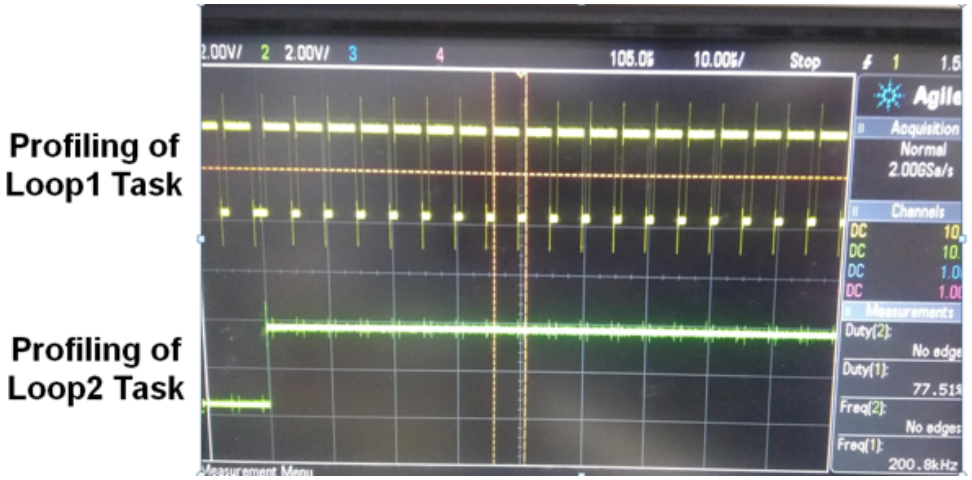

Figure 8. Profiling Waveforms for Both Tasks Running on C28x

Since the system is non-schedulable with C28x, one of the control tasks can be offloaded to CLA in order to meet the system requirements. As CLA offers very low interrupt latency, it is better to offload the fast control task to CLA, and this will also free up maximum bandwidth on C28x, which can be utilized for executing background and other system tasks. Figure 9 depicts the flow diagrams of both the tasks when the higher frequency Loop 1 task is offloaded to CLA. With the use of CLA for concurrent loop execution, the C28x utilization for control tasks has come down to approximately 7.7% allowing the other background task to execute correctly. Offloading the task to CLA makes the system perfectly schedulable in this case, which is also evident from the profiling waveforms shown in Figure 10. The example allows the user to offload the loop1 task quickly & conveniently from C28x to CLA by just updating the pre-defined symbol "run_loop1_cla" to 1 in the project build options.

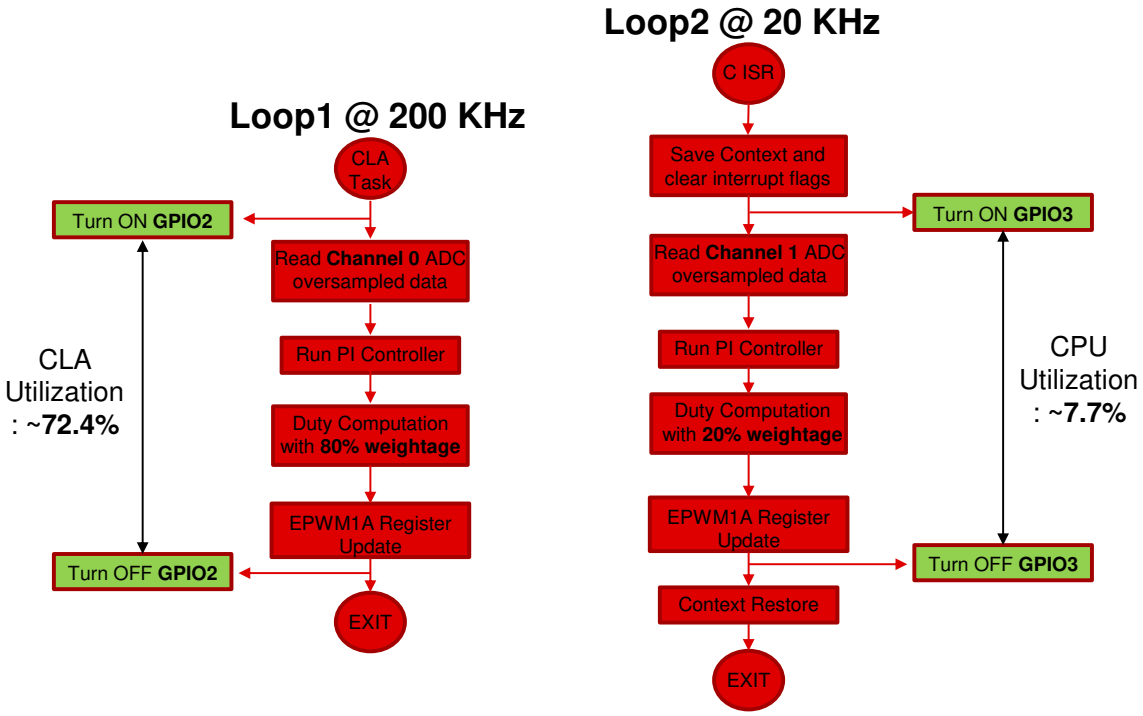

Figure 9. Flow Diagram With Loop 1 Task Offloaded to CLA

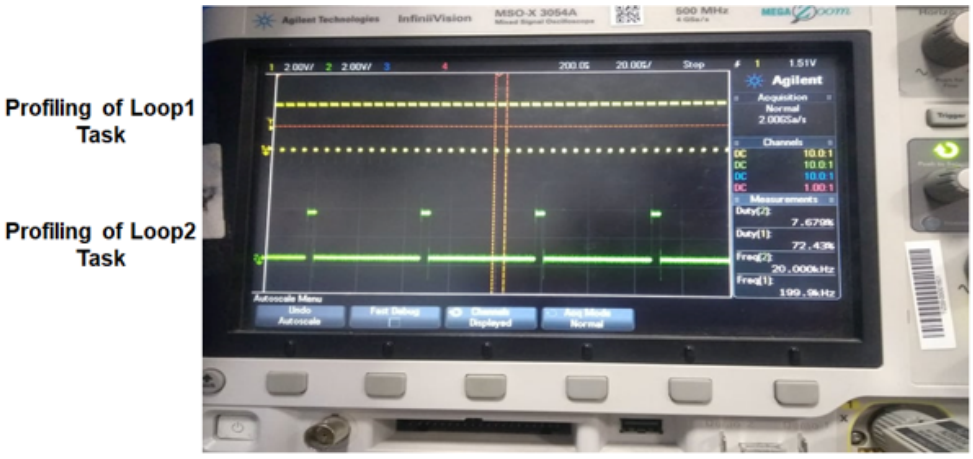

Figure 10. Profiling Waveforms for Loop1 Task Offloaded to CLA

5.1 Handling Shared Resources Across C28x/CLA

CLA allows offloading of control tasks efficiently from C28x and enables concurrent control loop execution on C2000 devices with many other additional benefits as discussed in earlier sections. But it is important to note that the peripherals are still shared between them and concurrent read-modify-write to the shared registers can lead to data race conditions ultimately leading to data violation or incorrect functionality. Ideally it's best to avoid any concurrent updates to the same peripheral by both CLA and C28x during run time but in case it is unavoidable, conflicts for shared resources must be handled carefully. The example "cla_ex7_shared_resource_handling" illustrates one such instance where both C28x and CLA do concurrent read-modify-write to same (AQCSFRC) register independently at different frequencies, which leads to a race condition between C28x and CLA and creates a possibility where updates due to one of them can get lost or overwritten. This is a standard critical section problem and can be handled using software handshaking mechanism like mutual exclusion but most of the real-time control applications are time-sensitive and cannot afford additional software cycles overhead. This example suggests an alternative hardware based technique to schedule the CLA and C28x tasks smartly in order to avoid overlapping access of shared resources. The hardware-based scheduling technique makes use of the programmable phase shifting mechanism of the EPWM modules.

As depicted in Figure 11, C28x ISR and CLA task runs independently at 10 KHz and 100 KHz respectively. C28x ISR gets periodically triggered by EPWM4, and toggles the EPWM1B output via software by controlling CSFB bits of AQCSFRC. CLA task gets triggered by EPWM5 and toggles the EPWM1A output via software by controlling CSFA bits of AQCSFRC (refer to device TRM [1] for further details about this register). Thus in this process both C28x and CLA do overlapping read-modify-write to AQCSFRC register as can be observed form the profiling waveforms shown in Figure 12. As a result, the updates to the AQCSFRC due to CLA gets overwritten, which is very evident from the spikes observed in EPWM1A output waveform shown in Figure 12.

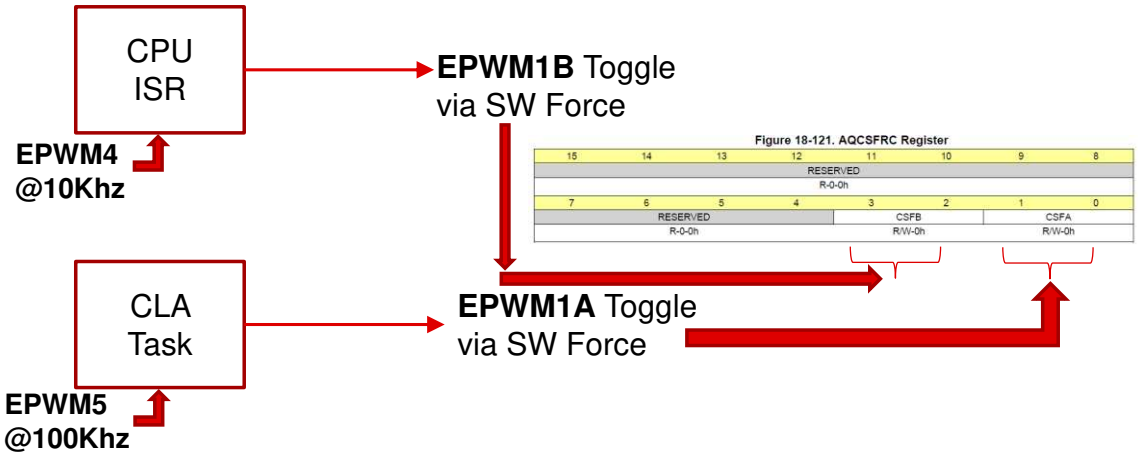

Figure 11. Concurrent R-M-W by C28x/CLA

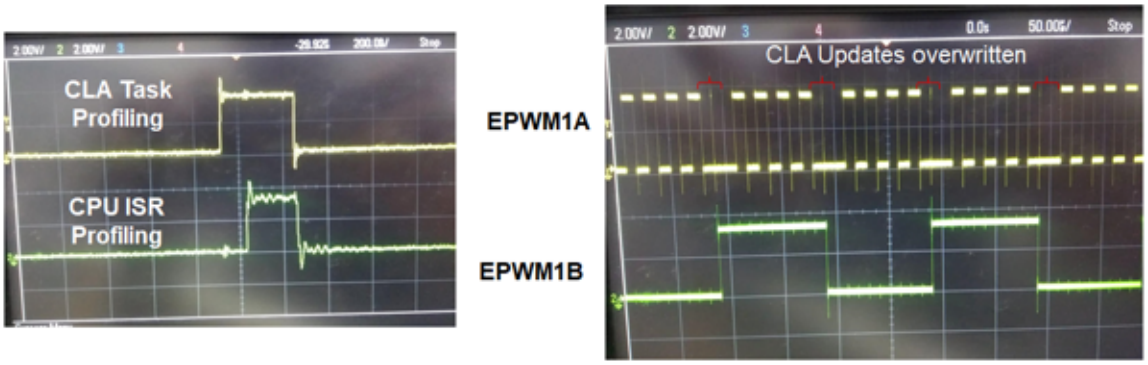

Figure 12. Profiling and Output Waveforms With Phase-Shift Disabled

The phase shifting mechanism of the EPWM modules, as shown in Figure 13, is utilized to schedule the CLA task and C28x ISR efficiently in order to resolve the above issue. EPWM4 generates a synchronous pulse every ZERO event and provides a phase shift of 20 cycles to EPWM5. This way both CLA task and C28x ISR runs at the original frequencies (100 KHz and 10 KHz), but CLA task leads with a phase offset of 20 cycles w.r.t C28x ISR as can be observed from the profiling waveform shown in Figure 14. Concurrent read-modify-writes to AQCSFRC never happens and the EPWM1A and EPWM1B outputs behave as desired without any distortion as shown in Figure 14. Thus the proposed hardware based scheduling technique helps to avoid data race conditions between C28x and CLA, and also helps to realize the true parallel execution of both processing engines by avoiding any simultaneous accesses and, hence maximizes the overall device performance.

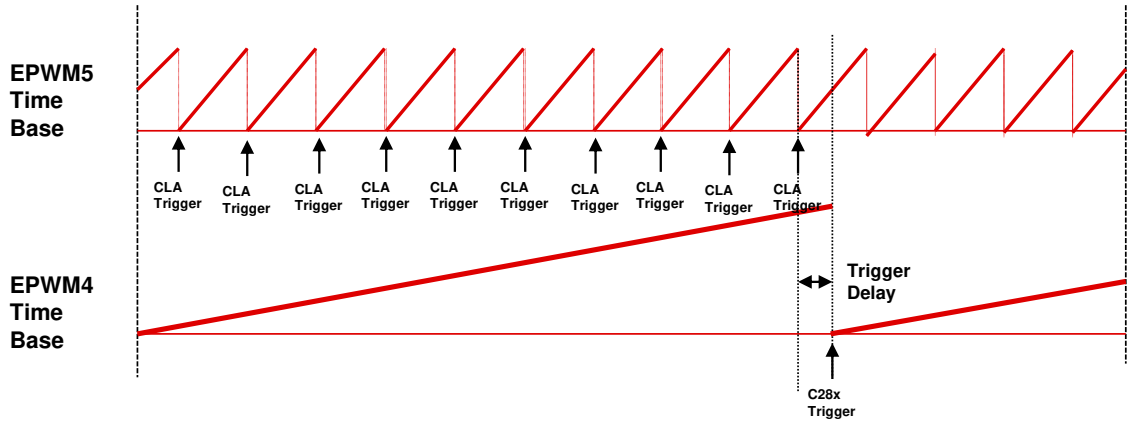

Figure 13. EPWM-Based Phase-Shifting Technique

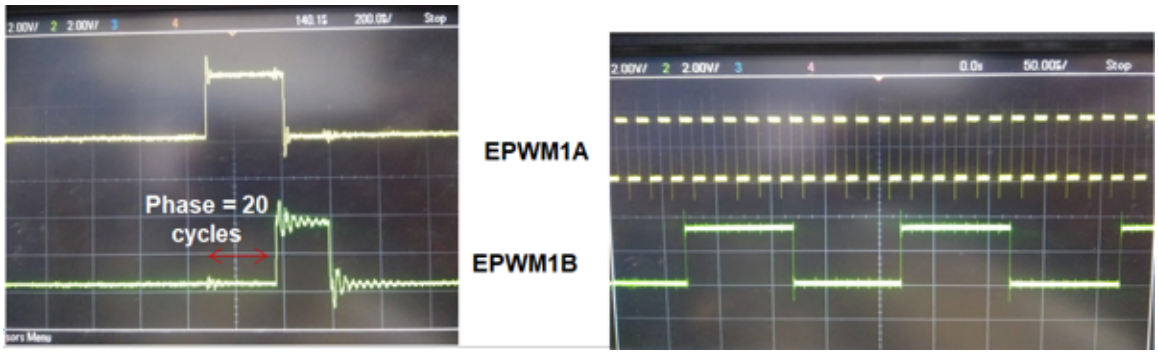

Figure 14. Profiling and Output Waveforms With Phase-Shift Enabled

6 Summary

The differentiation provided by TI's Control Law Accelerator (CLA) enables efficient execution of concurrent control loops on C2000 devices. CLA has been specially designed to boost the performance of control intensive math routines on real-time MCUs. The low-latency task-driven architecture of CLA is quite unique and reduces the sample-to-output delay, which is very critical for control applications. The direct access to key control peripherals and powerful floating-point processing capability allows CLA to offload the control tasks completely from the main CPU (C28x) thus freeing up its bandwidth to perform other system tasks. The CLA offers additional processing capabilities to C2000 devices and increases the overall device performance. The phase-shifting mechanism for scheduling CLA tasks as discussed in this report can be used to extract the maximum processing bandwidth out of the device. Another key benefit of the CLA over hardware-based control law implementations is flexibility. The CLA is fully programmable where developers can freely modify their control system without the time and high cost required to redesign a hardware-based solution. The C2000 C compiler [5] allows CLA to be programmed in C language similar to the C28x, which makes it very convenient to port existing algorithms or develop newer ones on CLA. The various software examples discussed in this application report demonstrates the key capabilities of CLA and can be used as a reference to adopt these unique features of CLA in their applications. These examples are very easy to use and do not require any special hardware platform other than the standard TI ControlCard to explore and evaluate the performance of CLA. Along with these examples, various Digital Power SDK [6] solutions also showcase the usage of CLA to reduce the overall C28x burden in various digital power solutions [7]. For further details on the software development and debugging with CLA, see [8].

7 References

1. Texas Instruments: *TMS320F28004x Microcontrollers Technical Reference Manual*
2. C2000Ware for C2000 MCUs
3. CLA Hands-On Workshop
4. Texas Instruments: *TMS320F2838x Microcontrollers With Connectivity Manager Data Sheet*
5. CLA C Compiler
6. Digital Power SDK for C2000 MCUs
7. CLA Usage in Valley Switching Boost Power Factor Correction (PFC) Reference Design
8. *C2000™ CLA Software Development Guide*

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265