C2000™ CLA Software Development Guide

1.0

Online HTML version available here

CONTENTS

1 Overview **1**

2 Getting Started **3**
- 2.1 Workshops . . . 3
- 2.2 Development Tools . . . 4
  - 2.2.1 Compiler and Assembly Tools . . . 4
  - 2.2.2 MathWorks Embedded Coder Support . . . 4
- 2.3 Examples . . . 4
- 2.4 FAQs and Debugging Tips . . . 5
- 2.5 System Use Cases . . . 5

3 Identifying Device-Specific CLA Features **6**
- 3.1 CLA Types . . . 6
- 3.2 Device-Specific Features . . . 8

4 Frequently Asked Questions **9**
- 4.1 Is the CLA independent from the C28x CPU? . . . 9
- 4.2 Can the C28x and CLA be synchronized? . . . 9
- 4.3 Is the CLA programmable? . . . 9
- 4.4 Is there a C compiler for the CLA? . . . 9
- 4.5 How is data shared between the CLA and C28x? . . . 10
- 4.6 How are data types different on C28x and CLA? . . . 10
- 4.7 What can trigger a task (also referred to as an ISR)? . . . 13
- 4.8 Is there support for nesting tasks? . . . 14
- 4.9 Is there a limitation on the code size for tasks? . . . 14
- 4.10 Which peripheral registers can the CLA access directly? . . . 15
- 4.11 Can the CLA send an interrupt to the C28x? . . . 16
- 4.12 Does the CLA have access to all memory blocks? . . . 16
- 4.13 If C28x and CLA can access the same device resource, which has priority? . . . 17
- 4.14 How can I measure the duration of a task? . . . 17
- 4.15 Can the C28x terminate a CLA task? . . . 18

4.16 What does the "Symbol, X, referenced in a.obj, assumes that data is blocked but is accessing non-blocked data in b.obj. Runtime failures may result" warning from the Linker mean? . . . 18

5 Debugging Tips **19**
5.1 Help! Why doesn't my CLA code work? . . . 20
5.2 My CLA task never starts. . . . 20
5.3 Why can't I force a task using software (IACK)? . . . 21
5.4 Why does my Code Composer Studio (CCS) GEL file enable the CLA clock on reset? . . . 21
5.5 Can I start a task from the debug window? . . . 21
5.6 After a "run", why did the CLA begin executing another task instead of stopping at MSTOP? . . . 21
5.7 Why are the variables in the CLA code not updating as expected? . . . 22

6 Comparison to C28x+FPU **23**
6.1 Are benchmarks for FPU div, sin, cos...etc. the same? . . . 23
6.2 Is the CLA floating-point multiply faster? . . . 24
6.3 What are the main differences? . . . 24

7 Support **26**

8 IMPORTANT NOTICE AND DISCLAIMER **27**

9 Changelog **28**

CHAPTER

ONE

OVERVIEW

The Control Law Accelerator (CLA) available on C2000™ MCUs is a fully-programmable independent 32-bit floating-point hardware accelerator that is designed for math intensive computations. The CLA can offer a significant boost to the performance of typical math functions that are commonly found in control algorithms.The CLA is designed to execute real-time control algorithms in parallel with the C28x CPU, effectively doubling the computational performance.This makes the CLA perfect for managing low-level control loops with higher cycle performance improvements over the C28x CPU. The CLA is also designed for fast interrupt response and has direct access to peripherals such as ePWM, HRPWM, eCAP, eQEP, ADC, DAC, CMPSS, PGA, SDFM, SPI, LIN, FSI, PMBus, CLB, and GPIO.

The CLA increases the overall compute bandwidth of the C2000 MCU and frees up the CPU for other tasks such as communications, systems, and diagnostics. Fig. 1.1 illustrates the key benefits of the CLA:

1. Reduced interrupt latency
2. Reduced control loop delay
3. Reduced C28x CPU bandwidth usage

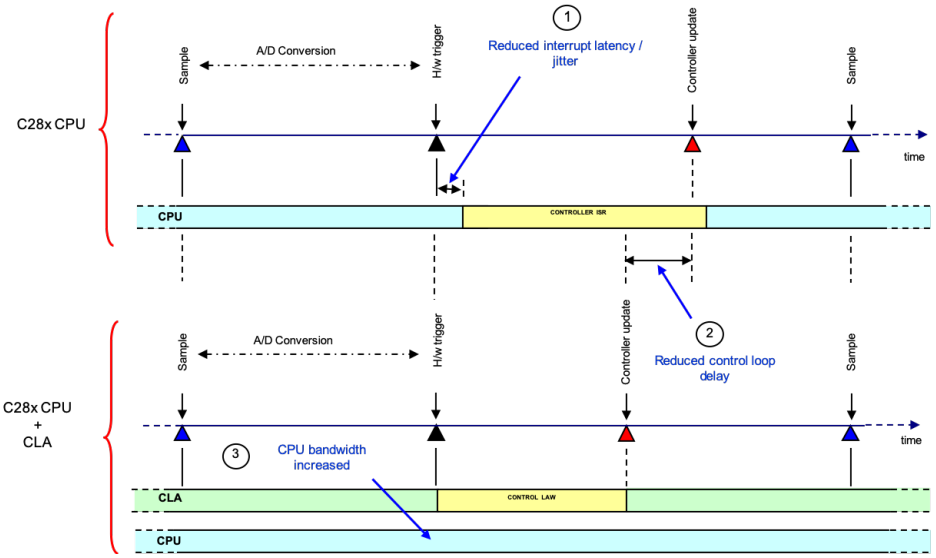


Fig. 1.1: Benefits of the CLA

Refer to these resources for more information on the CLA's benefits and its role in an overall system:

- Accelerators: Enhancing the Capabilities of the C2000™ MCU Family, section "Control Law Accelerator (CLA)".
- C2000™ Key Technology Guide
- CLA Usage in Valley Switching Boost Power Factor Correction (PFC) Reference Design

CHAPTER

TWO

GETTING STARTED

This section is a roadmap for getting started using various resources such as workshops, examples and system use cases. The recommended sequence is:

1. Study the *workshops*.
2. Experiment with *examples*.
3. Review the *Frequently Asked Questions* and *Debugging Tips*.
4. Refer to the *System Use Cases* for application specific examples of leveraging the CLA.

2.1 Workshops

- If you are new to C2000 MCUs, start with the C2000 Device Workshops
- Review the training material in the CLA Hands-On Workshop
- Review Chapter 9, Control Law Accelerator in the TMS320F28004x Microcontroller Workshop (pdf). Refer to C2000™ F28004x Microcontroller Workshop for the overall workshop.

Warning: New features may have been added to more recent devices or to the Compiler tools. Refer to the device-specific documents for the latest device (*Identifying Device-Specific CLA Features*) and compiler features (*Development Tools*).

Although the concepts taught in the workshops are applicable to all C2000 devices with the CLA, the projects may require older versions of the TI hardware and C2000 software. The workshop will list which hardware and software is supported.

2.2 Development Tools

The CLA is supported by TI's Code Composer Studio (CCS). Using CCS, both the C28x and CLA code can be developed and debugged within one integrated development environment.

2.2.1 Compiler and Assembly Tools

TI provides the CLA compiler as part of the C2000 Code Generation Tools. The CLA compiler and assembler are invoked automatically for CLA code. The Code Generation Tools are bundled in Code Composer Studio and can also be downloaded from: C2000 code generation tools - compiler.

CLA firmware can be written using C code or assembly. The recommendation is to use C code in your firmware as much as possible. If further optimizations are necessary, then assembly code can also be used.

Features of the CLA tools can be found in the following User's Guides:

- TMS320C28x Optimizing C/C++ Compiler User's Guide

  Describes the CLA compiler, its features and restrictions

- TMS320C28x Assembly Language Tools User's Guide

  Describes the extensions the assembler creates to support the CLA code. For example, command file sections and scratchpad sections.

2.2.2 MathWorks Embedded Coder Support

Visit the MathWorks website for information. MathWorks offers examples specifically for the CLA

2.3 Examples

1. Download the latest version of C2000Ware. For those new to C2000ware, refer to the C2000Ware chapter in the C2000™ Software Guide and this training video: Introduction to C2000Ware
2. Select an example project
   - Basic math examples: included with the CLAMath Library in \libraries\math\CLAmath
   - Device-specific examples: in the driverlib or device_support directories
3. Review concepts presented in the Hands on Workshop while working with the project
   - Configuration and CLA initialization performed by the C28x

- Set and remove breakpoint(s) in CLA code
- Single step CLA code and observe changes in memory or registers
- Observe how data is shared between the C28x and CLA (see also *How is data shared between the CLA and C28x?*)

2.4 FAQs and Debugging Tips

Refer to the *Frequently Asked Questions* for answers to commonly asked questions.

The *Debugging Tips* section provides help for common problems encountered when developing and debugging CLA code. Before using the information in this section, it is strongly recommended that you review the information in the CLA Hands-On Workshop.

2.5 System Use Cases

- The DigitalPower Software Development Kit (SDK) lists TI reference designs that leverage the CLA in the Features section.
- Refer to CLA Usage in Valley Switching Boost Power Factor Correction (PFC) Reference Design for a training video describing the usage of CLA in TIDM-1022.

CHAPTER

THREE

IDENTIFYING DEVICE-SPECIFIC CLA FEATURES

This chapter provides pointers to determine which features are implemented in the CLA available on a given device.

3.1 CLA Types

Table 3.1 lists describes the types of the CLAs and maps types to devices. The type of the CLA in a particular C2000 MCU is also indicated in the device datasheet.

Table 3.1: CLA Types

Type	Description	Devices
0	Original CLA module type	• F2803x (Only supports data RAM0 and 1 and does not allow CPU access when CLA dataRAM is enabled.) • F2805x, F2806x (Adds supports for data RAM2 and adds option to enable CPU access to data RAMs.)
1	• Increased Program address reachability to 16-bits • Added instructions to support the new address reach • Added two new offset addressing modes • CLA program memory is now user selectable and can reside anywhere in the lower 64k address space (excluding the M0 and M1 space). • Handing control to the CLA and assigning triggers to a task is now done at the system level. • A task can now fire an interrupt to main CPU mid execution.	F2807x, F2837xD, F2837xS
2	• Added background-code mode,that can run task like communications & clean-up routines in background • Background task runs continuously until disable or device/softreset • Background task can be triggered by a peripheral or software • Other foreground tasks can interrupt background task in the priority order defined • Added provision for making sections of background code uninterruptible • Added debug enhancements which has true software breakpoint support, where CLA re-fetches from the same address where halted during debug stop.	F28004x, F2838xD, F2838xS

3.2 Device-Specific Features

- **Device Technical Reference Manual (TRM)** Detailed CLA feature information for the specific device. This includes:
  - Configuration registers.
  - Pipeline.
  - Assembly instruction set.
- **Device Datasheet** Specifics related to the integration of the CLA into a device. This includes:
  - The device block diagram (bus connections).
  - Memories and peripherals accessible by the CLA (memory and register maps).
  - CLA frequency.

Note: To quickly find the device TRM and datasheet, browse to the product folder for your device. The product folder URL is of the form: **www.ti.com/product/<partnumber>**. Examples:

- www.ti.com/product/TMS320F28069
- www.ti.com/product/TMS320F28379D

The first document, at the top of the product folder, is the datasheet and the second is the TRM.

CHAPTER

FOUR

FREQUENTLY ASKED QUESTIONS

4.1 Is the CLA independent from the C28x CPU?

Yes. Once the CLA is configured by the main CPU it can execute algorithms independently. The CLA has its own bus structure, register set, pipeline and processing unit. In addition the CLA can access a host of peripheral registers directly. This makes it ideal for handling time-critical control loops and filtering or math algorithms.

4.2 Can the C28x and CLA be synchronized?

Synchronization is done by starting CLA execution either by a peripheral interrupt or by writing to a specific register. In addition the CLA can interrupt the main CPU.

4.3 Is the CLA programmable?

Yes. There are no algorithms built into the CLA hardware. The CLA is completely programmable in C or assembly.

4.4 Is there a C compiler for the CLA?

Yes. The TMS320C28x Code generation tools includes support for compiling CLA C code. Because of CLA architecture and programming environment constraints, the C language supported on the CLA has some restrictions. These are documented in the TMS320C28x Optimizing C/C++ Compiler User's Guide, Chapter "CLA Compiler".

4.5 How is data shared between the CLA and C28x?

The examples in C2000Ware demonstrate data exchange between the C28x and the CLA. This is done through RAM blocks that both of the CPUs can directly access. Since the CLA and the C28x code reside within same project, sharing data can be accomplished using these steps:

1. Create a shared header file with common constants and variables. Include this file in both the C28x C and CLA code.
2. Use data section pragma statements and the linker file to place the variables in the appropriate RAM.
3. Define shared variables in your C28x .c code. C28x and CLA shared global variables must be defined in the C28x .c code, and not the CLA code. The way data pages work on each device does not match. It is more constrained on the C28x side. Thus, data defined on the C28x side can be accessed on the CLA side, but not the other way around.
4. Initialize variables in the CPU to CLA message RAM with the main CPU.
5. Initialize variables in the CLA to CPU message RAM with a CLA task. This initialization task can be started via the main C28x software.

4.6 How are data types different on C28x and CLA?

See also:

C28x and CLA data types are documented in the version specific Compiler User's Guide. (*Compiler and Assembly Tools*)

To avoid ambiguity when sharing data between CLA and C28x, it is strongly recommended that you use type declarations that include size information. For example, don't use `int`. Use `int32_t` and `uint16_t` which are defined in `stdint.h` for both the C28x and CLA.

For example:

```
//
// in the shared header file and main .c file
// Does not indicate the size of the variable
// An unsigned int on C28x is 16-bit and an unsigned int on CLA is 32-
↪bits
//
unsigned int t0;
unsigned int t1;
//
// Instead use C99 data types with size information
//

```

```
12 #include <stdint.h>
13 uint16_t t0;
14 uint16_t t1;
```

Integers:

- For CLA an int is 32-bits
- For C28x an int is 16-bit

Pointers:

Pointers are interpreted differently on the C28x and the CLA. The C28x treats them as 32-bit data types (address bus size being 22-bits wide can only fit into a 32-bit data type) while the CLA only has an address bus size of 16 bits.

Assume the following structure is declared in a shared header file(i.e. common to the C28 and CLA) and defined and allocated to a memory section in a .c file:

Listing 4.1: Pointer Size Problem

```
      /
↪*************************************************************************
       shared.h
       C28x and CLA Shared Header File
     ␣
↪*************************************************************************/
      typedef struct{
        float a;
        float *b;
        float *c;
      }foo;


      /
↪*************************************************************************
       main.c
       Main C28x C-code Source File
     ␣
↪*************************************************************************/
      #pragma(X,"CpuToCla1MsgRam")         // Assign X to section␣
↪CpuToCla1MsgRam
      foo X;


      /
↪*************************************************************************
       test.cla
       CLA C-code Source File
```

```
↪**********************************************************************/
      __interrupt void Cla1Task1 ( void )
      {
        float f1,f2;
        f1 = *(X.b);
        f2 = *(X.c);     //Pointer incorrectly dereferenced
                         //Tries to access location 0x1503 instead
                         //of 0x1504
      }
```

Assume that the C28x compiler will allocate space for X at the top of the section CpuToCla1MsgRam as follows:

Element	Description	C28x Address
X.a	a is a 32-bit float in address	0x1500-0x1501
X.b	b is a 32-bit pointer in address	0x1502-0x1503
X.c	b is a 32-bit pointer in address	0x1504-0x1505

The CLA will interpret this structure differently. The CLA treats pointers ‘‘b’’ and ‘‘c’’ as 16-bits wide and, therefore, incorrectly dereferences pointer c.

Element	Description	CLA Address
X.a	a is a 32-bit float in address	0x1500-0x1501
X.b	b is a **16-bit** pointer in address	0x1502
X.c	c is a **16-bit** pointer in address	**0x1503**

The solution to this is to declare a new pointer as follows:

Create a new pointer ‘‘CLA_FPTR’’ which is a union of a 32-bit integer and a pointer to a float. The CLA compiler recognizes the size of the larger of the two elements (the 32 bit integer) and therefore aligns the pointer to the lower 16-bits. Now both the pointers ‘‘b’’ and ‘‘c’’ will occupy 32-bit memory spaces and any instruction that tries to de-reference pointer c will access the correct address 0x1504.

Listing 4.2: Pointer Size Solution

```
/**********************************************************************
  shared.h
  C28x and CLA Shared Header File
 **********************************************************************/
 typedef union{
   float *ptr; //Aligned to lower 16-bits
   Uint32 pad; //32-bits
 }CLA_FPTR;
```

```
typedef struct{
  float a;
  CLA_FPTR b;
  CLA_FPTR c;
}foo;


/*************************************************************************
 main.c
 Main C28x C-code Source File
*************************************************************************/
#pragma(X,"CpuToCla1MsgRam") //Assign X to section CpuToCla1MsgRam
foo X;



/*************************************************************************
 test.cla
 CLA C-code Source File
*************************************************************************/
__interrupt void Cla1Task1 ( void )
{
  float f1,f2;
  f1 = *(X.b.ptr);
  f2 = *(X.c.ptr); //Correct Access
}
```

4.7 What can trigger a task (also referred to as an ISR)?

The following sources can start (trigger) a task:

1. A peripheral interrupt
2. Software trigger
3. Another CLA task
4. On some devices a specific task can be designated as a background task. Refer to *Identifying Device-Specific CLA Features* for details.

The C28x configures the CLA. Part of the configuration is specifying what silicon resource trigger which tasks. There are generally two registers used to configure the triggers for a task.

- Older devices use the the MPISRCSEL1 register (i.e. 2803x, 2805x, 2806x).
- More recent devices use a system register CLA1TASKSRCSELx (2837xD/S, F28004x...).

See also:

- Examples in C2000Ware.
- The device-specific Technical Reference Manual (TRM) documents which peripherals can trigger which CLA tasks. (*Identifying Device-Specific CLA Features*).

The trigger can be an interrupt or it can be the C28x CPU through software. It is important to understand the trigger source is only the mechanism by which the task is started. The trigger source does not limit what the task can do or what registers it accesses.

- The main C28x CPU

  Can start a task by using the IACK #16bit instruction. For example IACK 0x0003 would flag interrupt 1 and interrupt 2. This is the same as setting bits in the force register (MIFRC)

- Another CLA task:

  On some devices, the CLA does not have access to directly force another task to be flagged. There are a couple of options:

  – Interrupt the C28x and force the task as part of the interrupt service routine.

  – Write to the ePWM register to force an interrupt which in-turn forces a task to be flagged.

4.8 Is there support for nesting tasks?

The CLA has its own fetch mechanism and can run and execute a task independently of the CPU. Only one task is serviced at a time - there is no nesting of tasks unless the background task in enabled. On devices with CLA Type 2 supporting background tasks, one level of nesting is possible. See *CLA Types*.

4.9 Is there a limitation on the code size for tasks?

Think of a task like an interrupt. It is a algorithm that is executed when the CLA receives an interrupt. The size of a task is only limited by the amount of program memory available and the CLA's program counter.

The start address of a task is configurable. Each task has an associated interrupt vector (MVECT1 to MVECT8). For type 0 CLAs, (*CLA Types*) this vector holds the starting address (as an offset from the first program location) of the task. For all the other types, this vector holds the entire 16-bit address of the task.

- CLA Type 0:

The program space is limited to 12-bits or 4096 words. All CLA instructions are 32-bits, so within a 4k x 16 program space you can have ~2k CLA instructions.

- CLA Type 1 and above:

  The program space is 16-bits wide, leaving the lower 64K words of space available as program space. Please refer to your device specific documentation for information on how to configure/allocate the memory spaces to the CLA.

The MSTOP instruction indicates the end of the task. After a task begins, the CLA will execute instructions until it encounters an "MSTOP" instruction.

4.10 Which peripheral registers can the CLA access directly?

Warning: Refer to the device-specific datasheet. If there is more than one CLA on a device, they may not all have the ability to connect to all of the same peripherals. (*Device-Specific Features*)

The answer is C2000 MCU family dependent. In general, more recently released device families support CLA access to more peripherals. The list below indicates where to look in the datasheet:

1. The device block diagram: indicates which peripherals are connected to the CLA bus.
2. The device level memory map: which memory blocks can be configured for CLA access
3. The external interface (EMIF) memory map: indicates any memory regions the CLA can access
4. The peripheral register memory maps: which specific peripheral registers the CLA has access to

Some examples:

- 2803x: Direct access to the ADC result, ePWM+HRPWM, and comparator registers.
- 2806x: Direct access to the ADC result, ePWM+HRPWM, eCAP, eQEP and comparator registers.
- 2807x: Direct access to the the ADC module(s) (including results), ePWM+HRPWM, eCAP, eQEP, comparator subsytem, DAC subsystem, SPI, McBSP, uPP, EMIF, GPIO
- 2837x: Direct access to the the ADC module(s) (including results), ePWM+HRPWM, eCAP, eQEP, comparator subsytem, DAC subsystem, SPI, McBSP, uPP, EMIF(s), GPIO

4.11 Can the CLA send an interrupt to the C28x?

Todo: check if there are new mechanisms on F28004x

Yes. The CLA can send an interrupt to the C28x as follows:

- End of Task:

  The CLA will send an interrupt to the PIE (peripheral interrupt expansion block) to let the main CPU (C28x) know a task has completed. Each task has an associated vector in the PIE. This interrupt is automatically fired when the associated task completes. For example when task 1 completes, CLA1_INT1 in the PIE will be flagged. If this interrupt is not enabled, then the C28x will ignore it.

- Forced Interrupt:

  This applies to CLA type 1 and later. (Note: when this option is enabled it automatically disables the end of task interrupt). The CLA has a Software Interrupt Capability, where a task can enable and force an interrupt to the main CPU. For example Task 1 can enable a software interrupt for task 2, by writing to the TASK2 bit of the CLA1SOFTINTEN register, and then forcing that interrupt by writing to the TASK2 bit of the CLA1SOFTINTFRC register.

- Overflow and Underflow:

  There are dedicated interrupts in the C28x PIE for CLA floating-point overflow and underflow conditions.

4.12 Does the CLA have access to all memory blocks?

No, the memory blocks that are available for use by the CLA is device family dependent.

See also:

The memory map in the device-specific datasheet indicates which blocks the CLA can access. (*Device-Specific Features*)

- The main CPU can allocate specific memory blocks to the CLA
- The memory map indicates which memory blocks can be configured for CLA usage.
- In some devices the memory usage (program or data) is fixed. In other devices some memory blocks accessible by the CLA can be configured as program memory or data memory.

In addition, there are dedicated message RAMs with a fixed configuration. Look for these in the memory map as well. Two examples are given below. Some devices have additional message RAMs.

- CLA to CPU Message RAM: CLA can read/write, main CPU can only read
- CPU to CLA Message RAM: CPU can read/write, CLA can only read

4.13 If C28x and CLA can access the same device resource, which has priority?

The device automatically arbitrates the access to the resource. That is, there is a defined order of priority.

See also:

Search for the word "arbitration" in the device-specific Technical Reference Manual (TRM) for more information. (*Device-Specific Features*).

Keep in mind if the main CPU performs a read-modify-write operation on a peripheral register and between the read and the write the CLA modifies the same register, the changes made by the CLA can be lost. In general it is best to not have both processors writing to the same registers.

If the CLA is configured to respond to a peripheral interrupt, then the interrupt will still go to the C28x PIE. This enables both CPUs to respond if needed.

4.14 How can I measure the duration of a task?

- Option 1: Use a timer:

  One method is to use a free running timer to capture the number of cycles the task execution took. The CLA has access to the PWM timers. As an example you could capture the number of cycles in C as follows:

  1. At the start of the task, CLA reads the timer
  2. Execute the task
  3. At the end of the task, CLA reads the timer again
  4. Calculate the number of cycles between (1) and (3)

```
ct1 = EPwm1Regs.TBCTR;
//
//..... code to be measured here .....
//
ct2 = EPwm1Regs.TBCTR;
delta_ct = ct2 - ct1;
```

- Option 2: Use a GPIO pin:

On some devices, the CLA has direct access to GPIO pins. In this case, instead of reading a timer, the CLA could toggle a pin before and after the code to be measured. Monitor the pin using an oscilloscope to measure the delta in time.

- Accounting for Overhead

  There will be some overhead due to triggering the CLA task and notifying the C28x that it has completed. In order to test this overhead, you could use a timer on the C28x to time the start and stop of the CLA task.

  1. Read the timer using the C28x
  2. Start the CLA task via the C28x Software
  3. Wait for the MIRUN bit to clear
  4. Read the timer using the C28x
  5. Calculate the delta between (1) and (4)
  6. Also refer to this post: https://e2e.ti.com/support/microcontrollers/c2000/f/171/t/738758

4.15 Can the C28x terminate a CLA task?

Yes. If a interrupt has been flagged, but the task has not yet run, the main CPU can clear the flag using the MICLR register.

If the task is already running then a soft reset (in MCTL) will terminate the task and clear the MIER register. If you want to clear all of the CLA registers you can use the hard reset option in the MCTL register.

4.16 What does the "Symbol, X, referenced in a.obj, assumes that data is blocked but is accessing non-blocked data in b.obj. Runtime failures may result" warning from the Linker mean?

This is a diagnostic added in C2000 code generation tools v20.2.x.LTS to detect when data accesses that assume data is blocked are made to non-blocked definitions.

The common case for this is a global CLA definition that is used in C28 code. For more background refer to 20.2.x LTS README entry for the new diagnostic.

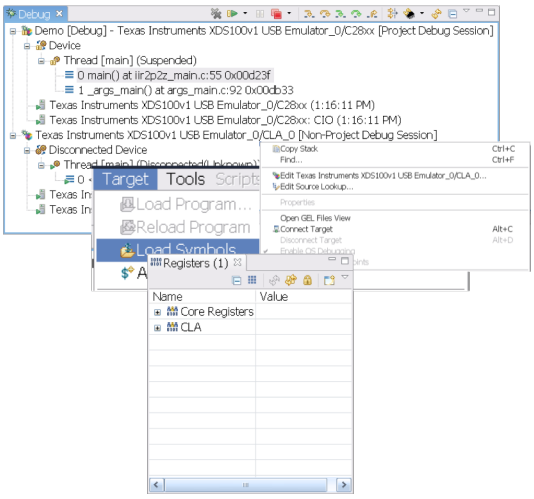
CHAPTER

FIVE

DEBUGGING TIPS

The CLA can halt, single-step and run independently from the main C28x CPU. Both the CLA and the C28x are debugged from the same JTAG port.

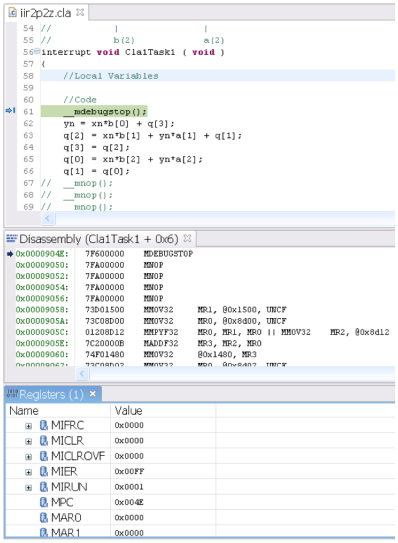
- When a debug session is launched, the debug view (window within CCS) will have entries for C28x and CLA.
- Clicking on CLA changes the context of all windows in CCS to reflect CLA data.
- Connecting to the CLA core from the debug perspective enables single stepping
- Load Program Symbols onto the CLA from Target menu options to begin debugging



Debugging CLA code

1. Insert a breakpoint into CLA code
   - An __mdebugstop() intrinsic is a CLA breakpoint. It will place the instruction MDEBUGSTOP at that position in assembly.
   - If not connected to the CLA core (i.e. single step disabled), the intrinsic behaves as a MNOP (no operation)

2. Start the task: CLA will execute code until the MDEBUGSTOP is in D2
3. Single step or run to the next breakpoint: Single stepping moves the CLA pipeline one cycle at a time



Note: The debugger cannot perform 'run to line' or 'step over' when debugging CLA code.

5.1 Help! Why doesn't my CLA code work?

Place a breakpoint in the CLA code. Make sure the CLA is connected and see if you hit the breakpoint. Step through the code to try and narrow down the problem.

If the breakpoint is never hit, then the task may not be starting. Check the CLA set-up (vectors, tasks are enabled, etc..)

5.2 My CLA task never starts.

Try starting the task with software to narrow down the issue. This will help confirm the task vector is setup correctly.

If the task fails to start for a peripherial trigger, check when the peripheral initalization is done. A CLA task only triggers on a level transition (an edge) of the configured interrupt source. If the peripheral triggers an interrupt before the CLA is initalized, then it can be missed. One way to get around this is to clear the interrupt flags from the peripherals.

5.3 Why can't I force a task using software (IACK)?

Verify the following:

1. This feature is enabled in the MCTL register. This register is EALLOW protected.
2. The interrupt is enabled in the MIER register. This register is EALLOW protected.
3. Software is configured as the task trigger. Refer to the device-specific TRM.
4. You are using the correct argument to IACK. For example IACK #0x0003 would flag interrupt 1 (bit 0) and interrupt 2 (bit 1).
5. Refer to the SW examples in C2000Ware.

5.4 Why does my Code Composer Studio (CCS) GEL file enable the CLA clock on reset?

When you reset the device, the CLA breakpoints are disabled and the CLA clock itself is disabled. To make debugging CLA code easier, CCS will watch for a reset condition and re-enable CLA breakpoints after a reset. For this to occur the CLA clock itself must be re-enabled. If you do not like this behavior, the line can be commented out or removed from the GEL file.

5.5 Can I start a task from the debug window?

You can use the CLA force register MIFRC to start tasks. This register is available in the CLA register window.

5.6 After a "run", why did the CLA begin executing another task instead of stopping at MSTOP?

If there is a task both pending and enabled when you run to the MSTOP, the pending task will automatically begin execution. To keep this from happening you can modify the MIER register so that no tasks are enabled.

5.7 Why are the variables in the CLA code not updating as expected?

A reason is that the `.scratchpad` and `.bss_cla` sections are incorrectly allocated to read only CLA Program memory. The task will run, but variables allocated to the `.bss_cla` section will not be updated. Moreover, the `.scratchpad` section is used by the compiler and allocating to to read only memory can result in undefined behavior.

CHAPTER

SIX

COMPARISON TO C28X+FPU

Note: This discussion is limited to the C28x + 32-bit FPU and does not include TMU or FPU64.

For this discussion, let us define the following instruction sets:

- **C28x Instruction Set** This is the original fixed-point instruction set.
- **C28x+FPU Instruction Set** This is the C28x Instruction Set plus additional instructions to support native single-precision (32-bit) floating-point operations. While the additional instructions are mostly to support single-precision floating-point math, there are some other useful instructions like RPTB (repeat block) included. Since they are part of the superset, and only available on devices with the FPU, we still refer to them as part of the FPU instructions.
- **CLA Instruction Set** The CLA instruction set is a subset of the FPU instructions. A few FPU instructions are not supported on CLA - for example the repeat block is not supported. The CLA also has a few instructions that the FPU does not have. For example: the CLA has some native integer math instructions as well as a native branch/call/return.

6.1 Are benchmarks for FPU div, sin, cos...etc. the same?

The CLA instructions are a subset of the C28x+FPU and for the math instructions there are equivalents on each, but there are still differences that impact benchmarks. For example:

- Cycle differences in multiply and conversion (see next question)
- Differences in branch and call instructions
- Resource differences (ex: 8 floating-point registers vs 4)
- Addressing modes

- CLA lacks RPTB (repeat block)

6.2 Is the CLA floating-point multiply faster?

Consider the following:

- C28x 32-bit FPU:

  Multiply or conversions take 2p cycles. This means that they take two cycles to complete, but remember you can put another instruction in that delay slot including another math instruction.

- CLA:

  The math instructions and conversions take 1 cycle - no delay slot needed. So if you were not able to use that delay cycle on the FPU to do meaningful work, then you could say the CLA is faster if you are just counting cycles.

6.3 What are the main differences?

The following table summarizes some of the main differences between the C28x + FPU and the CLA CPU at this time. Refer to the CLA documentation listed in the other resources for the latest information.

Table 6.1: CLA and C28x+FPU Comparison

Item	CLA	C28x+FPU
Execution	Independent, parallel execution with the C28x CPU	Part of the main C28x CPU
Floating-Point Registers	4 (MR0 - MR3)	8 (R0H - R7H)
Auxillary Registers	2 16-bits, (MAR0, MAR1) – Can access all of CLA data	8, 32-bits, (XAR0 - XAR7) – Shared with fixed-point instructions
Pipeline	8-stage pipeline – completely independent from C28x	8-stage pipeline – CPU fetch and decode shared with the fixed-point instructions
Single Step	Moves pipeline ahead 1 cycle	Completely flushes the pipeline
Addressing Modes	2: Direct and indirect with post increment. No data page pointer.	All C28x addressing modes
Interrupt Sources	Device dependent. Interrupts come directly to the CLA.	All available interrupts through the PIE.

Continued on next page

Table 6.1 – continued from previous page

Item	CLA	C28x+FPU
Nesting Interrupts	No stack pointer. CLA type 0 & 1: not supported. CLA Type 2: supports 1 background task	Nesting enabled through software
Instruction Set	Independent subset of FPU instructions. Similar mnemonics to C28x+FPU but with leading 'M' ex: MMPYF32 MR0, MR1, MR2.	Floating-point instructions are in addition(superset) to the C28x fixed-point instructions.
Repeated Instructions	No single repeat or repeat block	Repeat MACF32 & repeat block (RPTB)
Communication with C28x	Through shared RAM, message RAM, and interrupts. C28x can read CLA registers.	One CPU. Can copy between fixed and float registers
Math and Conversion	Single cycle	2p cycles (2 pipelined cycles)
Integer Operations	Limited support. Native instructions for AND, OR, XOR, ADD, SUB, shifts etc..	Uses fixed-point instructions
Flow Control	Native branch/call/return conditional delayed. 3 instructions before/after branch are always executed - performance can be improved by using delay cycles	Uses fixed-point flow control. Branches are not delayed – Instructions after are only executed if the branch is not taken Requires copy of float flags to fixed-point ST0
Memory Access	CLA program, data and message RAMs only. Refer to the memory map in the data manual.	All memory on the device
Register Access	Refer to the specific datasheet or TRM.	All peripherals on the device or specific C28x sub-system
Programming	CLA Assembly or CLA C Compiler. Requires C28x codegen 6.1.0 or later.	C or C++ or Assembly
Operating Frequency	Refer to the device datasheet	Refer to the device datasheet

CHAPTER

SEVEN

SUPPORT

To submit feedback on this guide or for C2000 related questions, please post to the C2000 microcontrollers forum in the TI E2E™ support forums.

CHAPTER

EIGHT

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), 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, 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 (https://www.ti.com/legal/termsofsale.html) 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.

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

CHAPTER

NINE

CHANGELOG

Table 9.1: Version History

Version	Date	Summary
v1.0	April 2020	Initial version of guide.