C2000™ Multicore Development User Guide

v1.0

Online HTML version available here

CONTENTS

This guide describes various cores available in C2000™ devices and how other resources such as peripherals, memories, GPIOs are shared among these. It also talks about how to communicate and share data among different cores. This guide is written for F2838x family of devices, but this can also apply for other devices with some exceptions.

CHAPTER

ONE

INTRODUCTION

Note: The online HTML version of this guide is available at

https://software-dl.ti.com/C2000/docs/C2000_Multicore_Development_User_Guide/index.html.

The F2838x family of devices contain 2 C28x CPU subsystems and one CM (Connectivity Manager) subsystem. The C28x CPU subsystem includes a C28x core, a CLA, and a DMA. The CM subsystem has an ARM Cortex M4 core and a uDMA.

Cores available in F2838x device:

- C28x1
- CPU1.CLA
- C28x2
- CPU2.CLA
- CM

Note that C28x1 and C28x2 cores are also referred as CPU1 and CPU2 respectively.

An important aspect in a multi-core application development is to split the resources across different cores. The F2838x device includes multiple memory blocks, peripherals and GPIOs, some of which are dedicated to certain cores, and some are shared across multiple cores. All the shared resources has to be assigned to an owner core before accessing the same. The ownership needs to be assigned by CPU1, which is the master core. Chapter *Ownership assignment of shared resources* provides details about the how to configure and assign ownership of various resources in the device.

Another important aspect is to enable effective communication and data sharing among various cores. The device has dedicated message RAMs which can be used for sharing data. It also includes an InterProcessor Communication (IPC) module which is used to communicate between C28x1, C28x2 and CM cores. This is further explained in chapters *Communication between CPU1, CPU2 and CM cores* and *Communication between C28x and CLA*.

The chapter *Debugging multiple cores* describes in detail how to load binaries in various cores and debug them. The chapter *Examples from C2000ware* demonstrates various examples available in C2000ware.

In the F2838x device, C28x1 core acts as the master core. The basic initialization of the device including the clocking and GPIO settings should be done by the CPU1 application. CPU1 can be booted in different modes based on the boot pin configuration. The CPU2 and the CM cores must be booted by the CPU1 application using the IPC module. Refer to the TRM chapter *ROM Code and Peripheral Booting* for more details on boot modes and boot pins. Some of these aspects are covered as part of example available in chapter *Examples from C2000ware*

For more details on CLA software development, please refer to to the C2000™ CLA Software Development Guide and the appnote Software Examples to Showcase Unique Capabilities of TI's C2000™ CLA

CHAPTER

TWO

MEMORY BLOCKS AVAILABLE

The following table lists the memories available in the device and their accessibility across cores. Please check the device datasheet for the actual size of the memories.

Table 2.1: Memory blocks available on F2838x

Memory	C28x	CLA	CM	Notes
Mx/Dx RAM	Y	N	N	Each CPU subsystem has dedicated RAM blocks.
LSx RAM	Y	Y	N	Each CPU subsystem has dedicated RAM blocks. Can be used as CLA data/program memory.
GSx RAM	Y	N	N	Shared between C28x1 and C28x2. Shared with DMA on each of the CPU subsystem.
Cx RAM	N	N	Y	
Sx/Ex RAM	N	N	Y	Shared with UDMA on CM subsystem.
CPUx-CPUx MSG RAM	Y	N	N	Separate RAM blocks for CPU1 to CPU2 and CPU2 to CPU1 communication.
CPUx-CM MSG RAM	Y	N	Y	Separate RAM blocks for CPUx to CM and CM to CPUx communication.
CLA Message RAM	Y	Y	N	Each CPU subsystem has dedicated RAM blocks. Separate RAM blocks for CPU to CLA and CLA to CPU communication.

continues on next page

Table 2.1 – continued from previous page

Memory	C28x	CLA	CM	Notes
CLA-DMA Message RAM	N	Y	N	Each CPU subsystem has dedicated RAM blocks. Separate RAM blocks for CLA to DMA and DMA to CLA communication.
Flash	Y	N	Y	CPU1, CPU2 and CM has dedicated flash memory blocks.

CHAPTER

THREE

OWNERSHIP ASSIGNMENT OF SHARED RESOURCES

3.1 LSxRAM

The dual-core C2000 device includes dedicated LS RAMs for CPU1 and CPU2 subsystems. LSxRAM can be owned by the C28x core or by the CLA core. The ownership of LS RAM should be assigned by the C28x core in the respective subsystem.

By default, all the LS RAMs are configured as C28x only memory.

Table 3.1: LSxRAM ownership assignment

Configuration	Register to be set	Driverlib function
C28x only memory	LSxMSEL.MSEL_LSx = 0	MemCfg_setLSRAMMasterSel (MEMCFG_SECT_LSx, MEMCFG_LSRAMMASTER_CPU_ONLY)
CLA program memory	LSxMSEL.MSEL_LSx = 1 LSxCLAPGM.CLAPGM_LSx = 1	MemCfg_setLSRAMMasterSel (MEMCFG_SECT_LSx, MEMCFG_LSRAMMASTER_CPU_CLA1) MemCfg_setCLAMemType (MEMCFG_SECT_LSx, MEMCFG_CLA_MEM_PROGRAM)
Data memory shared between C28x and CLA	LSxMSEL.MSEL_LSx = 1 LSxCLAPGM.CLAPGM_LSx = 0	MemCfg_setLSRAMMasterSel (MEMCFG_SECT_LSx, MEMCFG_LSRAMMASTER_CPU_CLA1) MemCfg_setCLAMemType (MEMCFG_SECT_LSx, MEMCFG_CLA_MEM_DATA)

3.2 GSx RAM

GS RAMs are shared between CPU1 and CPU2 subsystems. The ownership of GS RAM should be assigned by CPU1.

By default, all the GS RAMs are owned by CPU1 subsystem.

Table 3.2: GSxRAM ownership assignment

Configuration	Register to be set	Driverlib function
Owned by CPU1 subsystem	GSxMSEL.MSEL_GSx = 0	MemCfg_setGSRAMMasterSel (MEMCFG_SECT_GSx, MEMCFG_GSRAMMASTER_CPU1)
Owned by CPU2 subsystem	GSxMSEL.MSEL_GSx = 1	MemCfg_setGSRAMMasterSel (MEMCFG_SECT_GSx, MEMCFG_GSRAMMASTER_CPU2)

3.3 Peripherals

Most of the C28x peripherals are shared between CPU1 and CPU2. Please refer to the device datasheet for available peripherals.The ownership of the peripherals should be assigned by CPU1.

By default, all the peripherals are owned by CPU1.

Table 3.3: C28x Peripheral ownership assignment

Configuration	Register to be set	Driverlib function
Owned by CPU1	CPUSELx.module = 0	SysCtl_selectCPUForPeripheral (peripheral, instance, SYSCTL_CPUSEL_CPU1)
Owned by CPU2	CPUSELx.module = 1	SysCtl_selectCPUForPeripheral (peripheral, instance, SYSCTL_CPUSEL_CPU2)

There are few peripherals such as CAN, MCAN, Ethernet and USB which are shared across CPU1, CPU2 and CM cores. The ownership of shared peripheral should be assigned by CPU1.

Table 3.4: Shared Peripheral ownership assignment

Configuration	Register to be set	Driverlib function
Owned by CPU1	CPUSELx.bit.module = 0 PALLOCATE0.bit.module = 0	SysCtl_selectCPUForPeripheral (peripheral, instance, SYSCTL_CPUSEL_CPU1) SysCtl_allocateSharedPeripheral (peripheral, 0)

continues on next page

Table 3.4 – continued from previous page

Configuration	Register to be set	Driverlib function
Owned by CPU2	CPUSELx.bit.module = 1 PALLOCATE0.bit.module = 0	SysCtl_selectCPUForPeripheral (peripheral, instance, SYSCTL_CPUSEL_CPU2) SysCtl_allocateSharedPeripheral (peripheral, 0)
Owned by CM	PALLOCATE0.bit.module = 1	SysCtl_allocateSharedPeripheral (peripheral, 1)

Note: The peripheral clock must be enabled by the owner core.

3.4 GPIOs

The GPIOs on the devices are shared across all the cores and the ownership should be assigned by the CPU1 core before using them.

By default, all the GPIOs are owned by CPU1.

Table 3.5: GPIO ownership assignment

Configuration	Register to be set	Driverlib function
Owned by CPU1	GPxCSELy.bit.GPIOz = 0	GPIO_setMasterCore (pin, GPIO_CORE_CPU1)
Owned by CPU1.CLA	GPxCSELy.bit.GPIOz = 1	GPIO_setMasterCore (pin, GPIO_CORE_CPU1_CLA1)
Owned by CPU2	GPxCSELy.bit.GPIOz = 2	GPIO_setMasterCore (pin, GPIO_CORE_CPU2)
Owned by CPU2.CLA	GPxCSELy.bit.GPIOz = 3	GPIO_setMasterCore (pin, GPIO_CORE_CPU2_CLA1)
Owned by CM	GPxCSELy.bit.GPIOz = 4	GPIO_setMasterCore (pin, GPIO_CORE_CM)

CHAPTER

FOUR

COMMUNICATION BETWEEN CPU1, CPU2 AND CM CORES

4.1 Message RAMs

The device includes dedicated MSG RAMs for each combination of cores. For example, for sharing data from CPU1 to CPU2, CPU1 needs to write data to CPU1 TO CPU2 MSGRAM and CPU2 can read from this location. All the MSG RAMs are accessible to DMA (on C28x side) and to uDMA (on the CM side).

The following table lists out various Message RAMs available in the device and its accessibility across different cores. This also includes the name of the driverlib macro which holds the base address of the Message RAM.

- RW : Read and Write access
- R : Read access
- X : No access

Table 4.1: IPC Message RAMs

Memory	CPU1	CPU2	CM	Driverlib macro
CPU1_TO_CPU2 MSGRAM	RW	R	X	C28x: CPU1TOCPU2MSGRAM0_BASE CPU1TOCPU2MSGRAM1_BASE CM: NA
CPU2_TO_CPU1 MSGRAM	R	RW	X	C28x: CPU2TOCPU1MSGRAM0_BASE CPU2TOCPU1MSGRAM1_BASE CM: NA

continues on next page

Table 4.1 – continued from previous page

Memory	CPU1	CPU2	CM	Driverlib macro
CPU1_TO_CM MSGRAM	RW	X	R	C28x: CPUXTOCMMSGRAM0_BASE CPUXTOCMMSGRAM1_BASE CM: CPU1TOCMMSGRAM0_BASE CPU1TOCMMSGRAM1_BASE
CM_TO_CPU1 MSGRAM	R	X	RW	C28x: CMTOCPUXMSGRAM0_BASE CMTOCPUXMSGRAM1_BASE CM: CMTOCPU1MSGRAM0_BASE CMTOCPU1MSGRAM1_BASE
CPU2_TO_CM MSGRAM	X	RW	R	C28x: CPUXTOCMMSGRAM0_BASE CPUXTOCMMSGRAM1_BASE CM: CPU2TOCMMSGRAM0_BASE CPU2TOCMMSGRAM1_BASE
CM_TO_CPU2 MSGRAM	X	R	RW	C28x: CMTOCPUXMSGRAM0_BASE CMTOCPUXMSGRAM1_BASE CM: CMTOCPU2MSGRAM0_BASE CMTOCPU2MSGRAM1_BASE

Not that the addresses of the CPUx-CM MSG RAM on the C28x side and CM side are different. Also, on the C28x side, the addresses of CPU1_TO_CM and CPU2_O_CM MSG RAMs are the same. Depending on the core which is doing the write access (CPU1 or CPU2), the corresponding RAM block is updated. Similarly, same addresses are used for CM_TO_CPU1 and CM_TO_CPU2 MSG RAMs, depending on the core which is doing the read access (CPU1 or CPU2), the corresponding RAM block is read.

4.2 IPC Flags and Command registers

The device also includes Inter processor Communication (IPC) module for communication between cores. The F2838x device has 3 instances of IPC for the following communications:

- CPU1-CPU2
- CPU1-CM
- CPU2-CM

IPC includes registers for sending up to 32 flags from one core to another. For CPU1-CPU2 IPC instance, 4 of these flags have interrupt capability; which means, both C28x1 and C28x2 has 4 dedicated interrupt channels which can be triggered by the other C28x core. For CPUx-CM IPC instances, 8 out of the 32 flags have interrupt capability; which means, C28x1, C28x2 and CM has 8 dedicated interrupt channels which can be triggered by the other core.

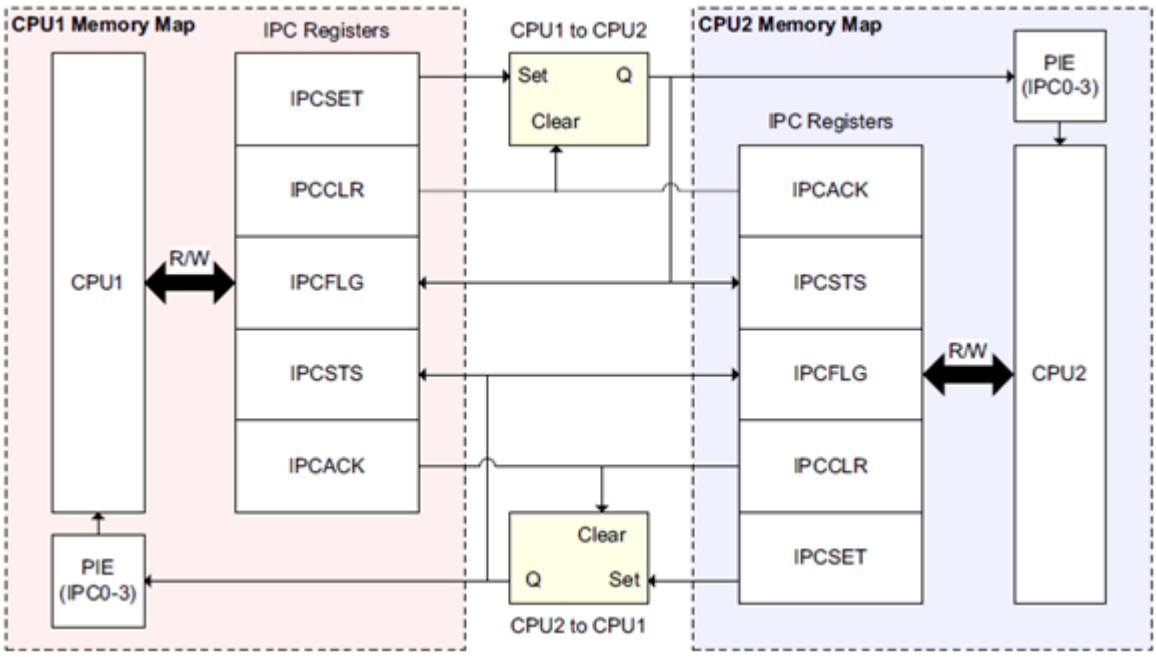


Fig. 4.1: IPC flags

Apart from the flags registers, the IPC module includes dedicated command registers. This include 4 registers for sending and 4 registers for receiving commands.

Local Register Name	Local CPU	Remote CPU	Remote Register Name
IPCSENDCOM	R/W	R	IPCRECVCOM
IPCSENDADDR	R/W	R	IPCRECVADDR
IPCSENDDATA	R/W	R	IPCRECVDATA
IPCREMOTEREPLY	R	R/W	IPCLOCALREPLY

Fig. 4.2: IPC command registers

For more details on the IPC flags and command registers, please refer to the TRM chapter *Interprocessor Communication (IPC)*.

Since the address spaces of CM and C28x are different, the driverlib functions used for sending and receiving messages includes a parameter *addrCorrEnable* which will correct the address while sending a command from CPUx to CM or vice versa. For more details, please refer to the driverlib API guide.

For sending data or commands from one core to the another, application can use the IPC flags, command registers and the message RAM as per the need. The image below shows a basic flow of how these can be used. The upcoming sections provide more details on how these are used for different usecases.

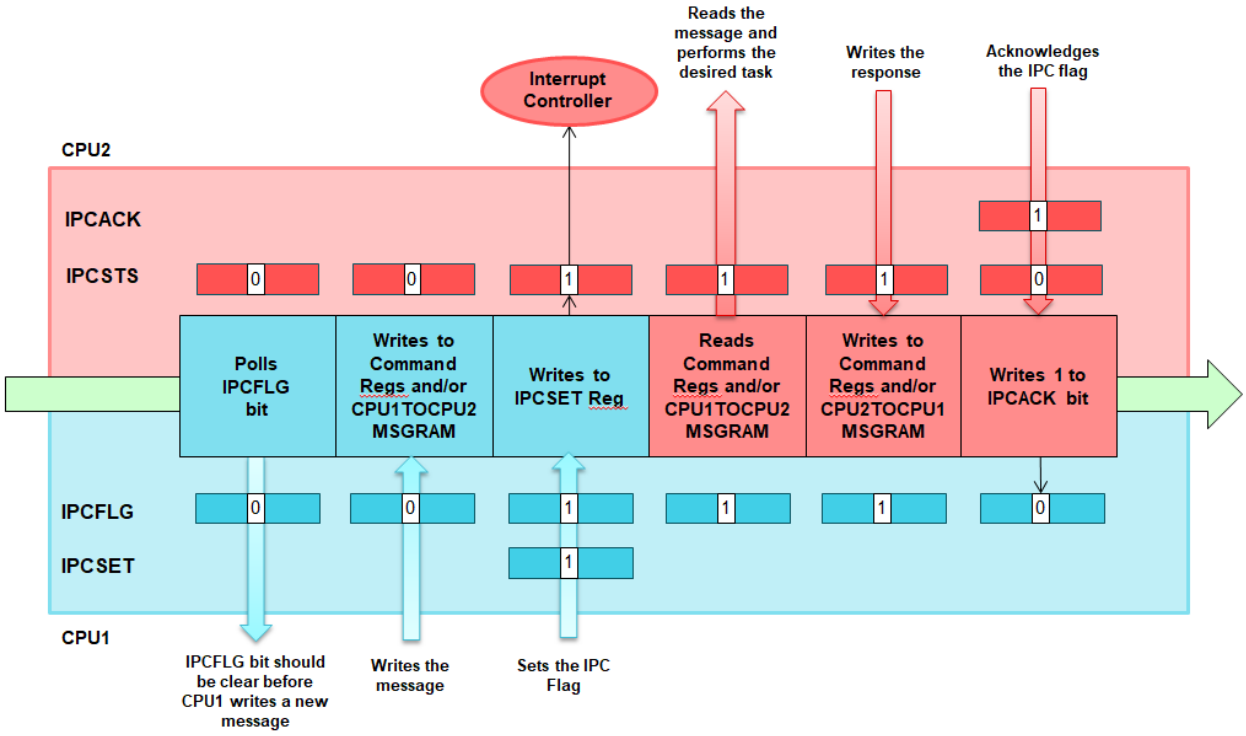


Fig. 4.3: IPC flow diagram

Note: The letters *L* and *R* used in IPC driverlib APIs denotes *Local* and *Remote* cores. For CPU1 to CPU2 communication, the enum *CPU1_L_CPU2_R* shall be used as the parameter *ipcType* in the driverlib functions.

4.3 Trigger an interrupt (with no data) from one core to another

1. Select a flag from the available IPC flags with interrupt capability. For CPU1-CPU2 IPC instance, IPC_FLAG0-3 has interrupt capability and for CPU1-CM and CPU2-CM IPC instances IPC_FLAG0-7 has interrupt capability.
2. **Core2 :** Enable interrupt corresponding the selected flag. Driverlib function: *IPC_registerInterrupt*.
3. **Core1 :** Make sure the corresponding bit in IPCFLG register is not already set. Driverlib function : *IPC_isFlagBusyLtoR*.
4. **Core1 :** Set the corresponding bit in IPCSET register. The bit in IPCFLG register will be set automatically. Driverlib function : *IPC_setFlagLtoR*.
5. **Core2 :** The corresponding bit in IPCSTS will be set and the interrupt is triggered.

6. **Core2 :** IPC ISR is invoked. Service the interrupt and acknowledge the flag by setting the IPCACK register. Driverlib function : *IPC_ackFlagRtoL.*
7. Setting the IPCACK register will clear the corresponding bit in IPCSTS register in Core2 and the IPCFLG register in Core1.

Core 1 can optionally wait for Ack from Core2 using the function *IPC_waitForAck*. If interrupt is not enabled on Core2, the IPC flag can be polled using the function *IPC_waitForFlag*.

4.4 Send a command from one core to another with interrupt

An IPC Command includes a 32 bit command, 32 bit address and a 32 bit data registers. Though the name suggests command, address and data, you can send any 3 32-bit data using the command registers.

1. Select a flag from the available IPC flags.
2. **Core2 :** Enable interrupt corresponding the selected flag. Driverlib function: *IPC_registerInterrupt*.
3. **Core1 :** Call the driverlib function *IPC_sendCommand*. The function checks if corresponding bit in IPCFLG register is not already set and sets the command, address and data registers to send to the the other core. And finally it sets the selected IPC flag.
4. **Core2 :** The corresponding bit in IPCSTS will be set and the interrupt is triggered.
5. **Core2 :** IPC ISR is invoked. Read the command registers and service the interrupt. Driverlib function : *IPC_readCommand*.
6. **Core2 :** Send response back to Core1 and acknowledge the flag by setting the IPCACK register. Driverlib functions : *IPC_sendResponse*, *IPC_ackFlagRtoL.*
7. Setting the IPCACK register will clear the corresponding bit in IPCSTS register in Core2 and the IPCFLG register in Core1.

Core 1 can optionally wait for Ack from Core2 using the function *IPC_waitForAck*. If interrupt is not enabled on Core2, the IPC flag can be polled using the function *IPC_waitForFlag*.

4.5 Sending a large amount of data from one core to another

1. **Core1 :** Write the data to the corresponding MSG RAM.
2. **Core1 :** Follow the steps mentioned above to send the command. The address parameter in the command can be set to address of the data in MSG RAM and data parameter can be set to the length of the data. *addrCorrEnable* should be set so that the function *IPC_sendCommand* corrects the address of the MSGRAM. This is important while sharing data between C28x and CM cores since the addresses of the MSG RAM is different in these cores.
3. **Core2 :** Follow the steps mentioned above to receive the command. The address and length of the data will be available in the received command.

4.6 Synchronize 2 cores

The IPC driverlib provides a function *IPC_sync* for synchronizing two cores. The function is expected to be called from both the cores using the same flag. It does the following in order:

- Sets the Flag from the Local core to Remote core
  - Sets the IPCSET register
  - This sets the IPSTS register in remote core
- Waits for the flag from Remote core to Local core
  - Waits for IPCSTS register
  - This is set when remote core sets the IPCSET register
- Acknowledges the flag from Remote core
  - Sets the IPC_ACK register
  - This clears the IPCFLG register in the remote core
- Waits for Acknowledge from the Remote core
  - Waits for IPC_FLG to be cleared
  - This is cleared when remote core writes to IPCACK register

This flow makes sure that neither core will return from this function before the other core enters it. That means, if Core1 reaches the sync function, it will wait until Core2 reaches the function and vice versa.

To use the IPC_sync function:

1. Select a flag from the available IPC flags. Preferably one without interrupt capability since we will be using polling method for synchronizing cores.
2. **Core1 :** Call *IPC_sync* function with the selected flag as parameter
3. **Core2 :** Call *IPC_sync* function with the selected flag as parameter

4.7 IPC Message queues

If you are using IPC command registers, queuing of commands is not possible, That means, core 1 can send a command only after the previous command is acknowledged. To overcome this, the IPC driver has implemented message queuing mechanism using software. It uses the MSG RAM to store and send the commands. The driverlib functions *IPC_initMessageQueue*, *IPC_sendMessageToQueue* and *IPC_readMessageFromQueue* can be used for this.

Please refer to the C2000ware driverlib example which showcases how to use the MSG RAMs, send IPC commands and trigger interrupt on the other core. It provides examples with and without using message queues. These examples are covered in *Examples from C2000ware* section.

CHAPTER

FIVE

COMMUNICATION BETWEEN C28X AND CLA

Each CPU subsystem contains a C28x core and a CLA core. C28x core can only communicate to the CLA core in its own subsystem. The IPC module is not available to CPU-CLA communication

For sharing data between C28x and CLA, CPU-CLA MSG RAMs can be used. The device includes dedicated MSG RAMs for CPU to CLA and CLA to CPU communication.

The following table lists out various CPU-CLA Message RAMs available in the device and its accessibility across different cores. This also includes the name of the driverlib macro which holds the base address of the Message RAM.

- RW : Read and Write access
- R : Read access

Table 5.1: CLA Message RAMs

Memory	C28x	CLA	Driverlib macro
CPU_TO_CLA MSG RAM	RW	R	CPUTOCLA_RAM_BASE
CLA_TO_CPU MSG RAM	R	RW	CLATOCPU_RAM_BASE

To send data from CPU to CLA, CPU needs to write the data in the CPUTOCLA_RAM which can be read by CLA and vice versa.

CLA tasks can be triggered by either C28x software or by peripheral events. C28x software can trigger CLA tasks by setting the corresponding bit in the MIFRC register. The driverlib function *CLA_forceTasks* does the same.

Note that the global variables defined in the .cla file are global to the cla source file, which means they are shared across CLA tasks, but not with the C28x core. All of the data shared between C28x and CLA must be defined in .c or .cpp file, and not in the .cla file.

Please refer to the c2000ware driverlib example which showcases how CPU triggers CLA tasks with CPU sending a data to CLA and CLA sending back the processed data. These examples are covered in *Examples from C2000ware* section.

CHAPTER

SIX

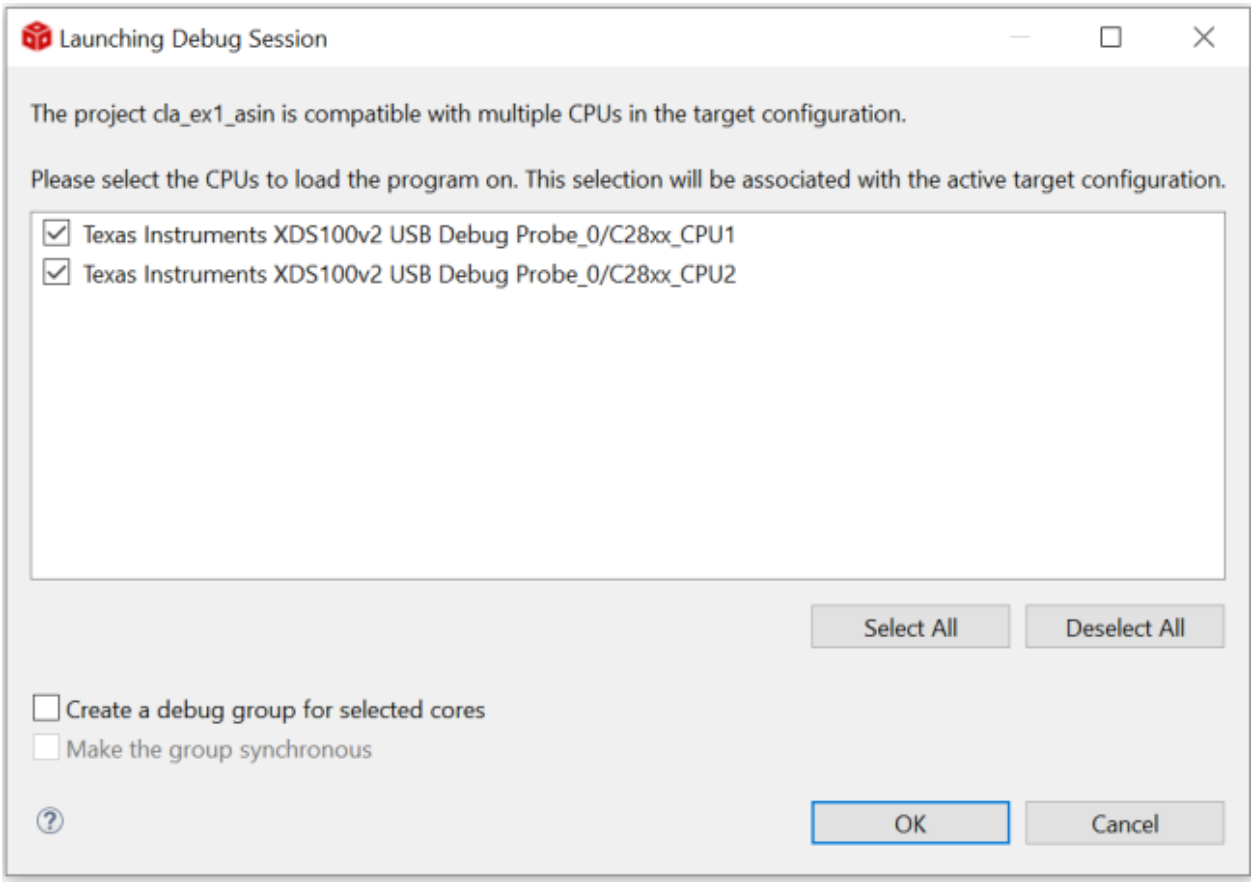
DEBUGGING MULTIPLE CORES

6.1 Loading program in multiple cores

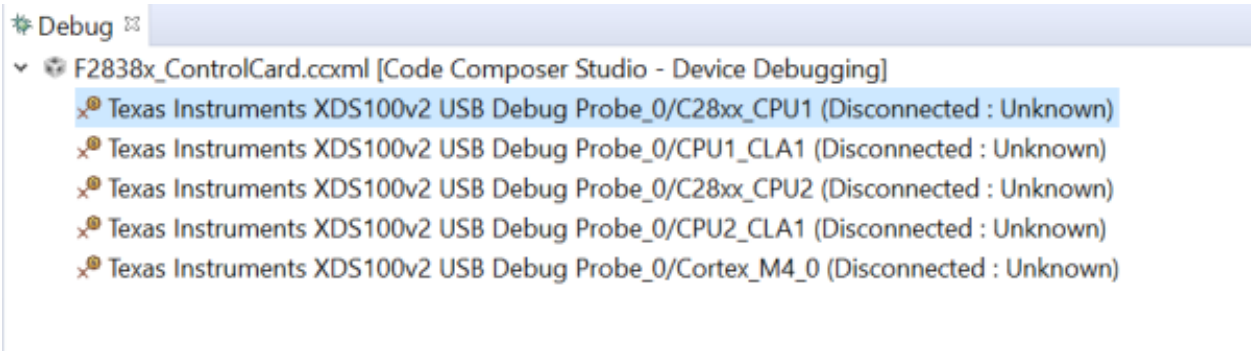
This section is applicable for CPU1, CPU2 and CM.

There are different ways to launch a debug session and load programs to various cores.

Project Debug : Once the CCS project is built, load the .out by selecting Run→Debug. CCS will identify compatible cores and prompt the user to select the required core(s). CCS will automatically connect to the selected cores and load the .out.



Project-less Debug Session : Under Target Configurations, right click on the required target configuration file and select 'Launch selected configuration'. Connect the required targets and load the .out by selecting Run->Load.

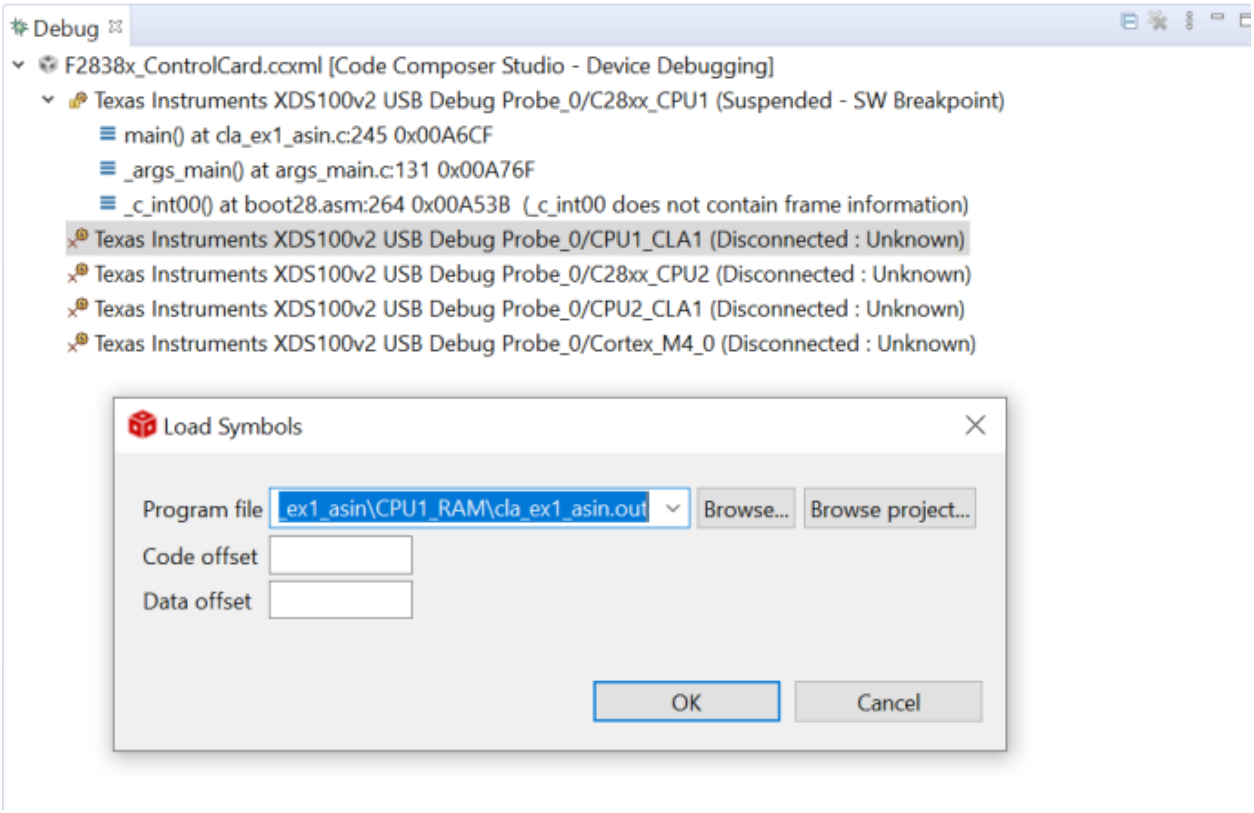


Note: The C28x1 is the master core in this device and must be connected and loaded prior to the other cores. The clocks and other basic system initialization must be done by the CPU1 application. For example, if you want to run a CM example, CPU1 should be loaded with a basic application that does the system initialization.

For more details on Multi-Core Debug in CCS : https://processors.wiki.ti.com/index.php/Multi-Core_Debug_with_CCS

6.2 Loading program in CLA

The CLA program is usually embedded in the corresponding C28x program itself and is loaded as part of loading the C28x core. Once the .out is loaded to C28x core, connect to the CLA core and execute 'Load symbols'. This option adds the symbols available in the .out for debugging purposes instead of loading the actual .out in the core.



The CLA program must be loaded in the LSxRAM. Note that the LSRAM must be configured as CLA program memory. In case of Flash configuration, the CLA program can be loaded in Flash, but must be copied to the LSxRAM. The linker command file should be updated to have this section load to Flash and run from LSxRAM. All the CLA data sections must be loaded into LSxRAM and RAM blocks must be configured as CLA Data RAM.

Please refer to the CLA example in C2000ware on memory configurations and linker command file updates needed.

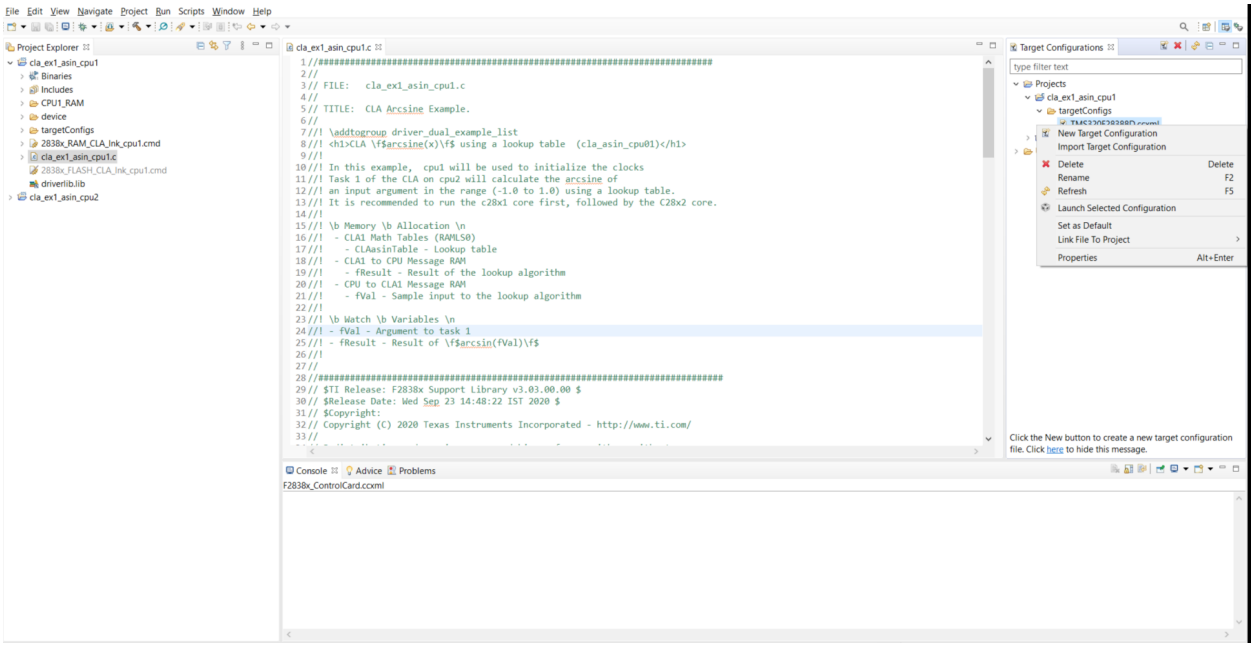
6.3 Exercise 1 - Multi-core Debug Example

Loading program in CPU1, CPU2, CPU2.CLA using the C2000ware example. The example is available in <C2000Ware>\driverlib\f2838x\examples\c28x_dual\cla\cla_ex1_asin.

This is a dual core example in which CPU1 application initializes the system clock and the CPU2 application triggers CLA tasks and shares some data to and fro. CPU2 application includes the CLA core code as well. In this exercise, we shall load both the applications in CPU1 and CPU2 cores respectively and Load symbols of CPU2 application in CPU2's CLA core.

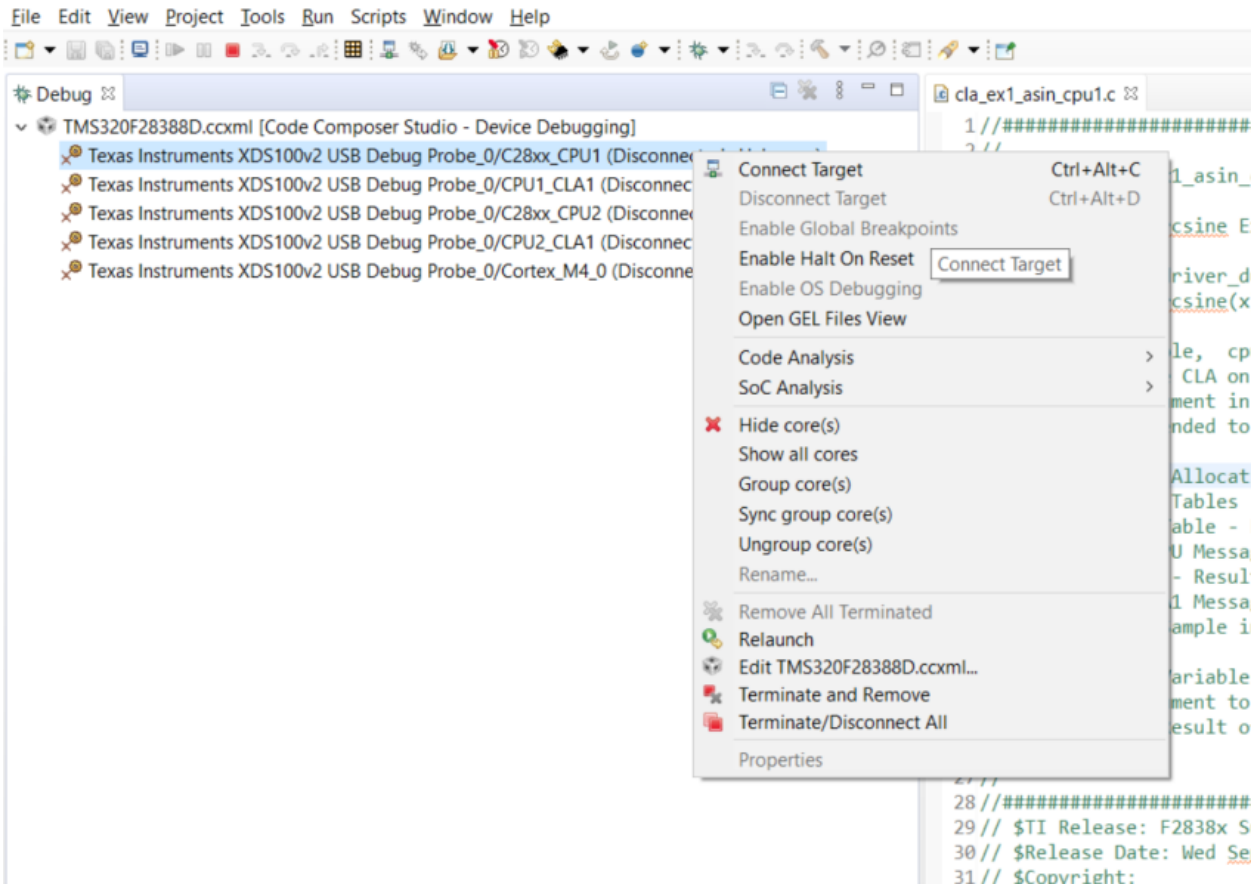
Step 1 : Launch the Target CCXML file.

Right click on the ccxml file in Target Configurations view -> Launch selected configuration.



Step 2 : Connect to CPU1 and load the CPU1 application.

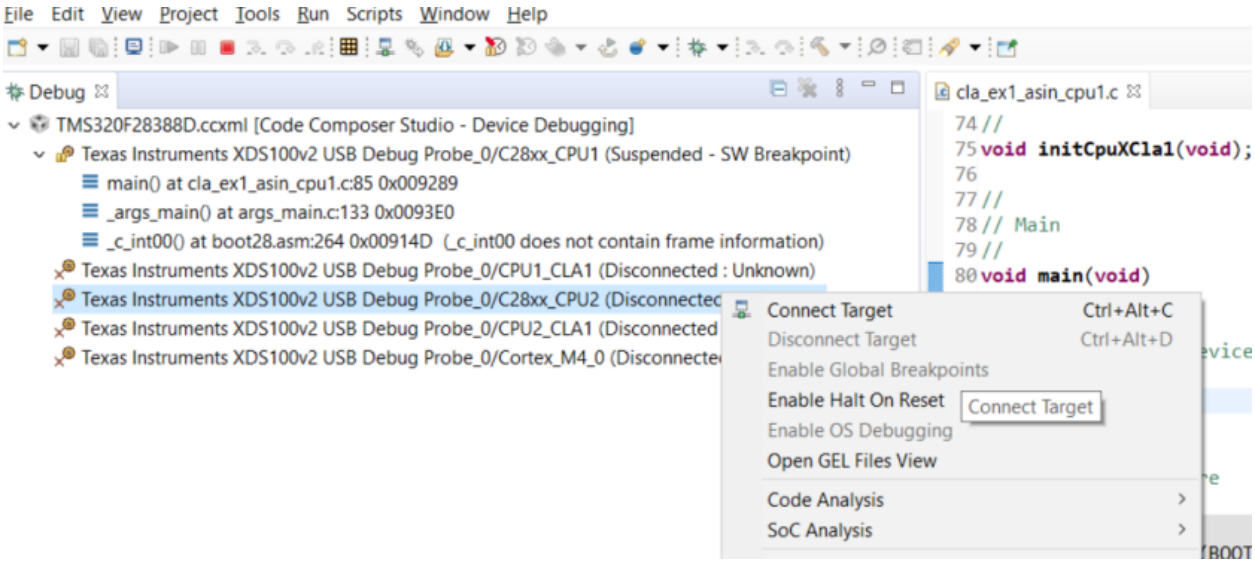
In the Debug view, right click on CPU1 core -> Connect Target.



Run -> Load -> Load Program and select the cla_ex1_asin_cpu1.out file.

Step 3 : Connect to CPU2 and load the CPU2 application.

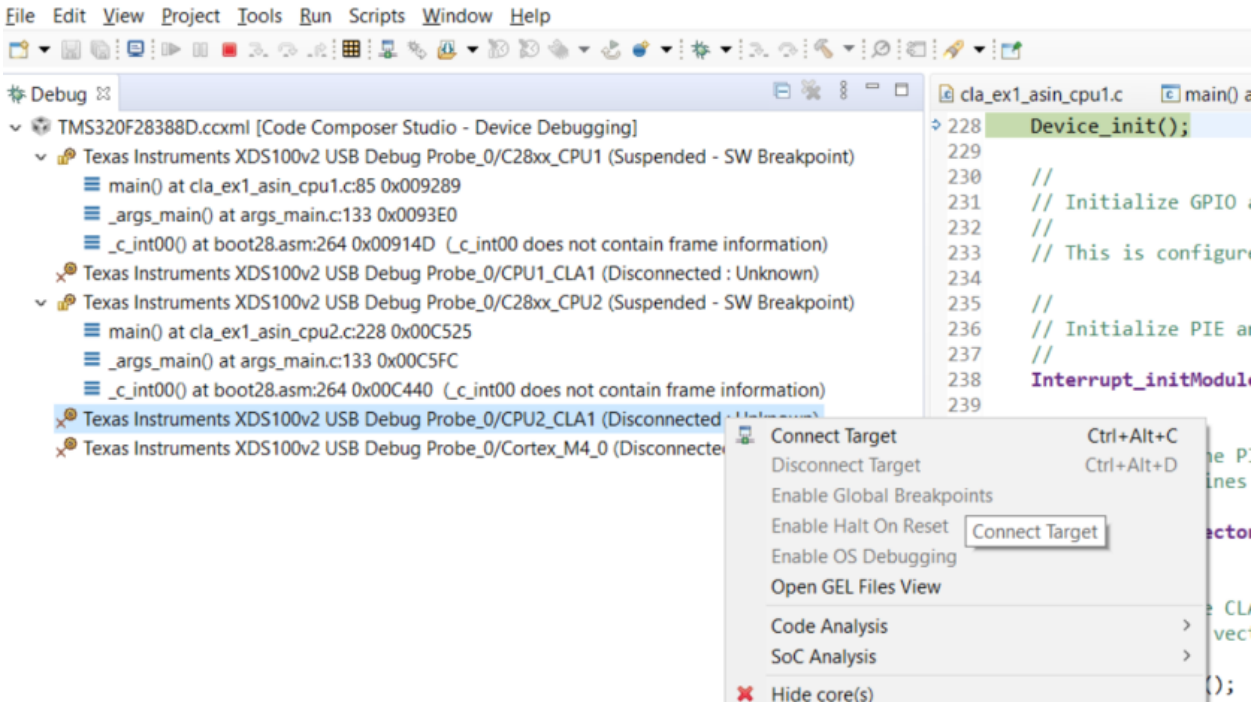
In the Debug view, right click on CPU2 core -> Connect Target.



Run -> Load -> Load Program and select the cla_ex1_asin_cpu2.out file.

Step 4 : Connect to CLA and load the CPU2 application symbols.

Since the CPU2 application is already loaded, we just need to Load the symbols of CPU2 application on the CLA core. In the Debug view, right click on CLA core -> Connect Target.



Run -> Load -> Load Symbols and select the cla_ex1_asin_cpu2.out file.

All the cores are currently in halted state. Select each core and click “Run” to execute the application. The CPU1 application should be run before running the CPU2 application. Note that this application has breakpoints inserted in the CLA code, and hence will be halted once the breakpoint is hit.

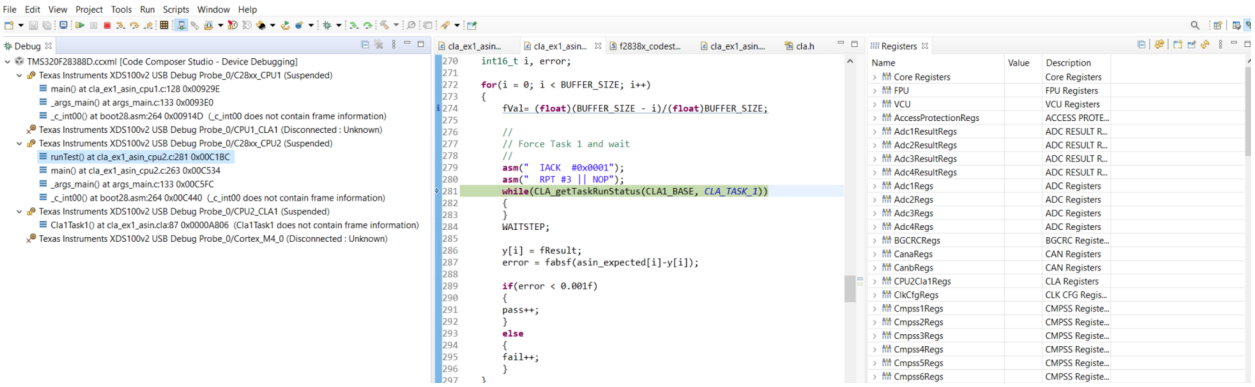
All the debugging features such as run, halt, step into, step over, breakpoints etc are available for all the cores. Make sure you select the correct core before using the controls.

6.4 Viewing Memories/Registers/Expressions

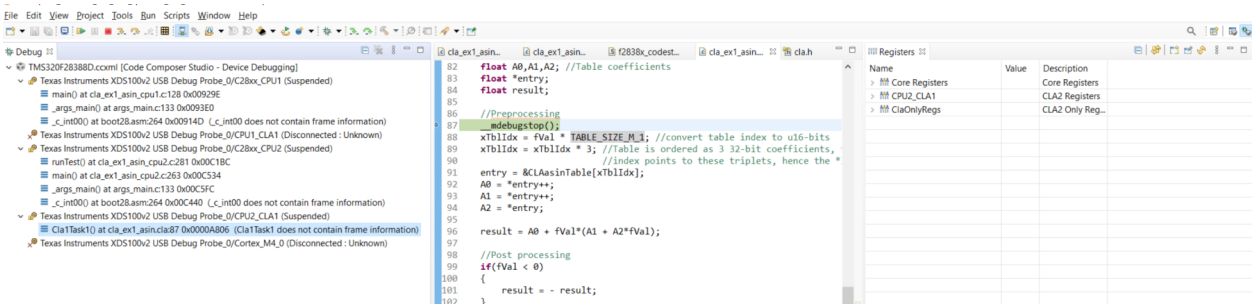
All the CCS views available for viewing the memory contents such as Registers, Memory Browser, Expressions, Variables, Disassembly etc are shared across cores. The contents will vary based on the active core.

6.5 Exercise 1 - Multi-core Debug Example (continued)

Click on the CPU2 core, the registers view displays all the registers available on the CPU2 core.



Click on the CLA core, the registers view displays all the registers available on the CLA core.



For more details on CLA debugging : https://training.ti.com/control-law-accelerator-cla-hands-workshop?context=1128629

CHAPTER

SEVEN

EXAMPLES FROM C2000WARE

This section goes through some of the examples available in C2000ware driverlib which demonstrates how the the cores within the device communicate with each and share data using Message RAMs. It also covers some examples which uses shared resources and how the ownership is assigned by the CPU1 master core.

7.1 Exercise 2 - IPC example

In this exercise, we will look into the an IPC example which showcases how to send a message from CPU1 to CPU2 with interrupts.

The example is available in <C2000ware>\driverlib\f2838x\examples\c28x_dual\ipc\CCS\ipc_ex1_basic.

- CPU1 application is sending a command to CPU2 along with a buffer of data. The buffer data is stored in the CPU1_TO_CPU2_MSGRAM.

```
#pragma DATA_SECTION(readData, "MSGRAM_CPU1_TO_CPU2")
uint32_t readData[10];
```

- The example uses the Device_bootCPU2 function to boot up the CPU2 code. Note that during debug time, CCS will boot up all the cores and having Device_bootCPU2 may not make much difference. This is needed for standalone (without debugger) execution of application.

```
#ifdef _FLASH
   Device_bootCPU2(BOOTMODE_BOOT_TO_FLASH_SECTOR0);
#else
   Device_bootCPU2(BOOTMODE_BOOT_TO_M0RAM);
#endif
```

- It uses the IPC_sync function to synchronize both the cores. This function will be returned only if the other core has also reached the IPC_sync function. This example uses Flag 31 for synchronizing.

  On CPU1 side:

```
IPC_sync(IPC_CPU1_L_CPU2_R, IPC_FLAG31);
```

  On CPU2 side:

```
IPC_sync(IPC_CPU2_L_CPU1_R, IPC_FLAG31);
```

- CPU1 uses FLAG0 for sending the command. An IPC command consists of a command, address and data. In this example, we are using a user defined command *IPC_CMD_READ_MEM*, base address of *readData* buffer as address and length of the *readData* as the data. CPU1 waits for acknowledgment from CPU2.

```
IPC_sendCommand(IPC_CPU1_L_CPU2_R, IPC_FLAG0, IPC_ADDR_CORRECTION_ENABLE,
                  IPC_CMD_READ_MEM, (uint32_t)readData, 10);

IPC_waitForAck(IPC_CPU1_L_CPU2_R, IPC_FLAG0);
```

- CPU2 enables the interrupt corresponding to Flag1. On receiving the command from CPU1, IPC_ISR0 ISR is invoked.

```
IPC_registerInterrupt(IPC_CPU2_L_CPU1_R, IPC_INT0, IPC_ISR0);
```

- In the IPC_ISR0 ISR function in CPU2, we read the command and the received memory location. CPU2 sends back a response and acknowledge the IPC Flag 0. The sending of response is optional, but acknowledging the flag is mandatory.

```
IPC_readCommand(IPC_CPU2_L_CPU1_R, IPC_FLAG0, IPC_ADDR_CORRECTION_ENABLE,
                  &command, &addr, &data);

...

IPC_sendResponse(IPC_CPU2_L_CPU1_R, TEST_PASS);

IPC_ackFlagRtoL(IPC_CPU2_L_CPU1_R, IPC_FLAG0);
```

- CPU1 on receiving the acknowledgement reads the response from CPU2.

```
IPC_getResponse(IPC_CPU1_L_CPU2_R);
```

Note that there is only one set of IPC command registers and 32 different flags. Sending another command using a different Flag will overwrite the contents of command registers. Hence it is recommended to send a command only after receiving the acknowledgment of the previous command.

Running the example :

- Build both the CCS projects and load there .outs in CPU1 and CPU2.
- Run CPU1 followed by CPU2.
- If the example runs as expected, the variable *pass* in CPU1 application will be updated with a value of 1.

The same example is available for CPU1-CM IPC, in <C2000ware>\driverlib\f2838x\examples\c28x_cm\ipc\CCS\ipc_ex1_basic. Note that in this case, we are using the CPU1_TO_CM_MSGRAM to store the *readData* buffer. The addresses used by CPU1 and CM to access the MSGRAM is different and hence it is very important to enable the address correction feature in *IPC_sendCommand* and *IPC_readCommand* function. Note that this should be used only if the *addr* parameter is an address in the IPC MSGRAM.

On CPU1 side:

```
IPC_sendCommand(IPC_CPU1_L_CM_R, IPC_FLAG0, IPC_ADDR_CORRECTION_ENABLE,
                  IPC_CMD_READ_MEM, (uint32_t)readData, 10);
```

On CM side:

```
IPC_readCommand(IPC_CM_L_CPU1_R, IPC_FLAG0, IPC_ADDR_CORRECTION_ENABLE,
                  &command, &addr, &data);
```

Note: Instead of using interrupts, the core can wait for a flag from the remote core using the function *IPC_waitForFlag*.

7.2 Exercise 3 - IPC example with message queues

In this exercise, we will look into the an IPC example which showcases how to send a message from CPU1 to CPU2 with interrupts using message queue.

The example is available in <C2000ware>\driverlib\f2838x\examples\c28x_dual\ipc\CCS\ipc_ex2_msgqueue.

As mentioned in the previous exercise, there is only one set of command registers and hence queuing of messages is not possible with IPC command registers. The IPC driver implements a message queuing mechanism to address this issue. This implements circular buffers in the MSG RAM to store the messages instead of the IPC command registers. For every IPC instance, there are 4 buffers implemented with a size of 4 messages per buffer. These 4 buffers are tied to the IPC flags 0 to 3. In this example CPU1 sends a message to CPU2 and expects a message back.

- CPU1 application is sending a command to CPU2 along with a buffer of data. The buffer data is stored in the CPU1_TO_CPU2_MSGRAM.

```
#pragma DATA_SECTION(readData, "MSGRAM_CPU1_TO_CPU2")
uint32_t readData[10];
```

- In this example we are using the buffers tied to IPC FLAG1 for both CPU1 to CPU2 and CPU2 to CPU1 communication.

  CPU1 side :

```
IPC_initMessageQueue(IPC_CPU1_L_CPU2_R, &msqQ, IPC_INT1, IPC_INT1);
```

  CPU2 side :

```
IPC_initMessageQueue(IPC_CPU2_L_CPU1_R, &msqQ, IPC_INT1, IPC_INT1);
```

- As mentioned in the previous exercise, this example uses Device_bootCPU2 to boot the CPU2 core and uses IPC_sync function to synchronize both the cores. Note that the example uses the sync function after initializing the message queues on both the cores.
- CPU1 creates a TX message instance which consists of a command, address and 2 data. In this example, we are using a user defined command *IPC_CMD_READ_MEM*, base address of *readData* buffer as address and length of the *readData* as the data1, a message identifier as data2. After sending the message, it waits for a message from CPU2 in the same message queue.

```
TxMsg.command = IPC_CMD_READ_MEM;
TxMsg.address = (uint32_t)readData;
TxMsg.dataw1  = 10;  // Using dataw1 as data length
TxMsg.dataw2  = 1;   // Message identifier

IPC_sendMessageToQueue(IPC_CPU1_L_CPU2_R, &msqQ,
                       IPC_ADDR_CORRECTION_ENABLE,
                       &TxMsg, IPC_BLOCKING_CALL);

IPC_readMessageFromQueue(IPC_CPU1_L_CPU2_R, &msqQ,
                         IPC_ADDR_CORRECTION_DISABLE,
                         &RxMsg, IPC_BLOCKING_CALL);
```

- CPU2 enables the interrupt corresponding to Flag1. On receiving the message from CPU1, IPC_ISR1 ISR is invoked.

```
IPC_registerInterrupt(IPC_CPU2_L_CPU1_R, IPC_INT1, IPC_ISR1);
```

- In the IPC_ISR1 ISR function in CPU2, we read the message and the received memory location. CPU2 sends back a message using the message queue and acknowledge the IPC Flag 0. The sending of response is optional, but acknowledging the flag is mandatory.

```
IPC_readMessageFromQueue(IPC_CPU2_L_CPU1_R, &msqQ,
                         IPC_ADDR_CORRECTION_ENABLE,
                         &RxMsg, IPC_NONBLOCKING_CALL);

IPC_sendMessageToQueue(IPC_CPU2_L_CPU1_R, &msqQ,
                       IPC_ADDR_CORRECTION_DISABLE,
                       &TxMsg, IPC_NONBLOCKING_CALL);

IPC_ackFlagRtoL(IPC_CPU2_L_CPU1_R, IPC_FLAG1);
```

Running the example :

- Build both the CCS projects and load there .outs in CPU1 and CPU2.
- Run CPU1 followed by CPU2.
- If the example runs as expected, the variable *pass* in CPU1 application will be updated with a value of 1.

The same example is available for CPU1-CM IPC, in <C2000ware>\driverlib\f2838x\examples\c28x_cm\ipc\CCS\ipc_ex2_msgqueue. Note that in this case, we are using the CPU1_TO_CM_MSGRAM to store the *readData* buffer. The addresses used by CPU1 and CM to access the MSGRAM is different and hence it is very important to enable the address correction feature in *IPC_sendMessageToQueue* and *IPC_readMessageFromQueue* function. Note that this should be used only if the *addr* parameter is an address in the IPC MSGRAM.

```
IPC_sendMessageToQueue(IPC_CPU1_L_CM_R, &msqQ, IPC_ADDR_CORRECTION_ENABLE,
                       &TxMsg, IPC_BLOCKING_CALL);

IPC_readMessageFromQueue(IPC_CM_L_CPU1_R, &msqQ, IPC_ADDR_CORRECTION_ENABLE,
                     &RxMsg, IPC_NONBLOCKING_CALL);
```

7.3 Exercise 4 - CLA example

In this exercise, we will look into a CLA example which showcases how to share data between C28x and CLA cores.

The example is available in <C2000ware>\driverlib\f2838x\examples\c28x\cla\CCS\cla_ex1_asin.

In this example, C28x core shares a data to CLA and triggers a CLA task to compute its asin value. The CLA returns the computed value back to C28x core.

- CPU1 application is sending a data to CLA and receiving back a data from CLA. The data to be sent, *fVal* variable, is stored in CPU_TO_CLA_MSGRAM and the data to be received, *fResult* variable is stored in CLA_TO_CPU_MSGRAM. Both these variables are defined in .c file.

```
#pragma DATA_SECTION(fVal, "CpuToCla1MsgRAM")
float fVal;
#pragma DATA_SECTION(fResult, "Cla1ToCpuMsgRAM")
float fResult;
```

- In this example, LS5RAM is used as CLA program memory and LS0RAM and LS1RAM is used as CLA data memory. CPU1 configures these memories accordingly.

```
MemCfg_setLSRAMMasterSel(MEMCFG_SECT_LS5, MEMCFG_LSRAMMASTER_CPU_CLA1);
MemCfg_setCLAMemType(MEMCFG_SECT_LS5, MEMCFG_CLA_MEM_PROGRAM);
```

(continues on next page)

(continued from previous page)

```
MemCfg_setLSRAMMasterSel(MEMCFG_SECT_LS0, MEMCFG_LSRAMMASTER_CPU_CLA1);
MemCfg_setCLAMemType(MEMCFG_SECT_LS0, MEMCFG_CLA_MEM_DATA);

MemCfg_setLSRAMMasterSel(MEMCFG_SECT_LS1, MEMCFG_LSRAMMASTER_CPU_CLA1);
MemCfg_setCLAMemType(MEMCFG_SECT_LS1, MEMCFG_CLA_MEM_DATA);
```

- CLA Task 1 is used to compute the asin value of *fVal* variable and update *fResult* variable. CPU1 maps the CLA task function and enables tasks and interrupts. *Cla1Task1* is CLA task function which is written in .cla file and cla1Isr1 is the ISR function on the C28x core which gets invoked once the CLA task is completed.

```
CLA_mapTaskVector(CLA1_BASE,CLA_MVECT_1, (uint16_t)&Cla1Task1);

CLA_enableIACK(CLA1_BASE);
CLA_enableTasks(CLA1_BASE, CLA_TASKFLAG_ALL);

Interrupt_register(INT_CLA1_1, &cla1Isr1);
Interrupt_enable(INT_CLA1_1);
```

- CPU1 updates the *fVal* variable an triggers CLA task 1 and reads the *fResult* variable after a delay. Instead of a constant delay, CPU1 can also wait for the CLA run status using the driverlib function *CLA_getTaskRunStatus*.

```
fVal = (float)(BUFFER_SIZE - i)/(float)BUFFER_SIZE;
CLA_forceTasks(CLA1_BASE,CLA_TASKFLAG_1);
WAITSTEP;
y[i] = fResult;
```

- CLA can use the variables *fVal* and *fResult* like normal global variables.

```
extern float fVal;
extern float fResult;
...

__interrupt void Cla1Task1 ( void )
{
   int xTblIdx;
   float result;
   ...
   xTblIdx = fVal * TABLE_SIZE_M_1;
   ...
   fResult = result;
}
```

Similar example is available for CPU2 in <C2000ware>\driverlib\f2838x\examples\c28x_dual\cla\CCS\cla_ex1_asin.

Running the example :

- Build the CCS project and load the .out in CPU1.
- If you need to debug the CLA code or view memories, load symbols of the same .out to CLA core. This step is not mandatory for running the example.
- If the example runs as expected, the variable *pass* will be updated with a value of 64. You could also monitor the values of *fVal* and *fResult* in each test iteration.

7.4 Exercise 5 - Dual core GPIO example

In this exercise, we will look into the LED blinky example on the CPU1 and CM cores. In this example, 2 GPIOs which are tied to LEDs on the controlcard are used - one is controlled by the CPU1 core and the other is controlled by the CM core.

The example is available in <C2000ware>\driverlib\f2838x\examples\c28x_cm\led\CCS\led_ex1_c28x_cm_blinky.

- As described in the previous exercises, CPU1 boots up CM.

```
#ifdef _FLASH
   Device_bootCM(BOOTMODE_BOOT_TO_FLASH_SECTOR0);
#else
   Device_bootCM(BOOTMODE_BOOT_TO_S0RAM);
#endif
```

- The GPIO configuration registers are only accessible by CPU1. Hence both the pins are configured by CPU1 application.

```
Device_initGPIO();
GPIO_setPadConfig(DEVICE_GPIO_PIN_LED1, GPIO_PIN_TYPE_STD);
GPIO_setDirectionMode(DEVICE_GPIO_PIN_LED1, GPIO_DIR_MODE_OUT);
GPIO_setPadConfig(DEVICE_GPIO_PIN_LED2, GPIO_PIN_TYPE_STD);
GPIO_setDirectionMode(DEVICE_GPIO_PIN_LED2, GPIO_DIR_MODE_OUT);
```

- By default, all the GPIOs are owned by CPU1. CPU1 assigns the ownership of LED2 pin to CM.

```
GPIO_setMasterCore(DEVICE_GPIO_PIN_LED2, GPIO_CORE_CM);
```

- CPU1 updates the LED1 pin and CM updates the LED2 using the *GPIO_writePin* or *GPIO_togglePin* functions.

Running the example :

- Build both the CCS projects and load the .outs in CPU1 and CM cores.
- Run CPU1 core followed by CM core.
- If the example runs as expected, you will see the 2 LEDs on the controlcard blinking.

Similar example is available for CPU1 and CPU2 in <C2000ware>\driverlib\f2838x\examples\c28x_dual\led\CCS\led_ex1_c28x_dual_b

7.5 Exercise 6 - CM example with shared peripheral

In this exercise, we will look into CAN examples on the CM core. As mentioned earlier, CAN is shared across CPU1, CPU2 and CM cores. Even though this is a CM-only example, we need an application running on the CPU1 core (master core) which does the basic initialization of the device.

The example is available in <C2000ware>\driverlib\f2838x\examples\cm\can and the CPU1 application is available in <C2000ware>\driverlib\f2838x\examples\cm\can\CCS\can_config_c28x.

- CPU1 does the basic initialization of the device such as setting up the clock, GPIO initialization and booting of CM core.

```
Device_init();
Device_bootCM(bootmode);
Device_initGPIO();
```

- Configure the GPIOs required for CAN. Since in this example, GPIOs are not controlled by GPIO Data registers, assigning the master core is not necessary.

```
GPIO_setPinConfig(GPIO_36_CANA_RX);
GPIO_setPinConfig(GPIO_37_CANA_TX);
```

- Assign the ownership of CAN to CM core.

```
SysCtl_allocateSharedPeripheral(SYSCTL_PALLOCATE_CAN_A,0x1U);
```

Running the example :

- Build the can_config_c28x project and load to CPU1.
- Build the required CAN example and load to CM core.
- Run CPU1 core followed by CM core.
- Refer to the selected example in CM for the expected outcome.

CHAPTER

EIGHT

FREQUENTLY ASKED QUESTIONS

- **How to enable standalone boot of different cores?**
  - CPU1 is booted as per the boot pin configuration.
  - CPU2 and CM needs to be booted by CPU1. This can be done by using the driverlib functions *Device_bootCPU2* and *Device_bootCM*
  - CLA does not need explicit boot. Once the CLA clocks are enabled, the CLA tasks will get executed on receiving the respective triggers.

  Please refer to the device TRM chapter *ROM Code and Peripheral Booting* for more details.
- **Does having a breakpoint in CM halt CPU1?**

  No, the breakpoints are separately available in different cores (CPUx, CPUx.CLA and CM). If any core hits a breakpoint, none of the other cores will be halted.
- **CPU2 is unable to write to a particular register or memory location. What could be the reason?**

  Some of the memories and peripherals are shared across multiple cores. Make sure the ownership of the memory / peripheral is correctly assigned before accessing them.

  There are certain register blocks which are only accessible by the CPU1 core. Please refer to the TRM register descriptions to know the accessibility of registers from different cores.
- **Why am I unable to get CM to toggle a GPIO pin?**

  The GPIOs are shared across cores. Make sure the ownership of the GPIO is correctly assigned before accessing them. The basic GPIO configuration such as direction, pull-up, polarity, open-drain and so on must be done by the CPU1 core.
- **Can I use GSRAM as stack memory or for storing global variables?**

  Yes, GSRAM can be used as stack memory and also for storing global variables. This can be done by updating the linker command files. But note that these are shared by CPU1 and CPU2 and be careful not to use the same memories by both the cores. In case of CPU2, CPU1 needs to assign the ownership of the GSRAM to CPU2.

  On contrast, the LSRAMs are not shared and are physically different memories even though the addresses used by CPU1 and CPU2 are the same.
- **Can I use GSRAM to store CLA data /program?**

  No, only LSRAMs are accessible by CLA.
- **Getting the error "Program will not fit into available memory" for Cla1Prog section.**

  This means the the CLA program size is more than the assigned memory block. You can map multiple memory blocks to the Cla1Prog section. For example, if the original configuration was :

```
RAMLS4    : origin = 0x00A000, length = 0x000800
RAMLS5    : origin = 0x00A800, length = 0x000800

Cla1Prog : > RAMLS4
```

There are 2 ways to map RAMLS5 as well to Cla1Prog section.

Option 1:

```
RAMLS4    : origin = 0x00A000, length = 0x000800
RAMLS5    : origin = 0x00A800, length = 0x000800

Cla1Prog : >> RAMLS4 | RAMLS5
```

Option 2:

```
/* RAMLS4    : origin = 0x00A000, length = 0x000800 */
/* RAMLS5    : origin = 0x00A800, length = 0x000800 */
RAMLS4_5    : origin = 0x00A000, length = 0x001000

Cla1Prog : > RAMLS4_5
```

In Option 1, Cla1Prog section will be split into 2 subsections and mapped to LS4 and LS5 memories. In option 2, the LS4 and LS5 are combined to form a bigger memory block which is used to map the Cla1Prog section.

Note that in both cases, both RAMLS4 and RAMLS5 should be configured as CLA program memory.

- **Why is "Load Symbols" needed for the CLA core?**

  The CLA code is actually embedded in the respective C28x application. When the .out is loaded to the C28x core, the CLA application as well gets loaded. To add the debug information on the CLA core, you need to load the symbols of the .out. This step does not load the actual .out file. This will just load the debug symbols available in the .out file to the CLA core. Even if this step is skipped, the CLA will run the tasks as expected. But when you connect to the CLA core, the source code and variables/expression will not be visible.

CHAPTER

NINE

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