Application Note

Dual-Core Debugging on the F28P65x Device

Ira Thete, Mithra Kandasamy

ABSTRACT

The C28x family of microcontrollers consists of many dual-core devices. These may be two identical or unidentical cores. The key application areas of such devices are industrial drives, solar inverters, power storage etc. Each of the core handles different functions in the application. For example, in industrial drives the first core can be used to handle the inner current/speed control loop, and the second core can be used to deal with I/O communication and fault coordination. This application note explores how to program and debug a dual-core C28x device and the common mistakes to avoid.

Table of Contents

Trademarks

1 Introduction

When it comes to real-time control applications, there is always a fundamental concern. Deterministic hard-real-time control loops (typically running at frequencies in the order of 100 kHz) cannot share compute time with slower supervisory tasks such as communications, protection logic, telemetry, fault management etc. A second CPU alleviates this concern by dividing the overall workload and allowing for a single CPU's dedicated bandwidth to real-time-control related tasks. This application note explores how to effectively debug applications that might show unexpected behaviors in a dual core environment. It talks about how boot up sequence and memory or peripherals ownership conflicts can lead to the erratic performance of a dual core application.

2 Dual Core Architecture in C28x family

The C2000 dual-core lineup spans two architectural flavors:

2.1 Homogenous Architecture (C28x + C28x)

Both CPUs are identical C28x DSP cores running the same ISA, the same compiler tool-chain, and the same instruction set extensions such as FPU and TMU. The two cores are symmetric in capability – either one can run any task. Some examples of such devices are

- TMS320F2837x
- TMS320F28P65xx

Even though the hardware is symmetric, the cores are not functioning the same way during the startup. CPU1 is hardwired as the system master CPU.

2.2 Heterogenous Architecture (C28x + C28X + ARM Cortex M4)

In this flavor, the dual core C28x subsystem handles real-time control as before. The ARM Cortex0M4 "Connectivity Manager" is a general-purpose 32-bit RISC core running at 125 MHz, purpose-built for running communication middleware and RTOS-based tasks.

2.2.1 TMS320F2838x (F28388D, F28386D, etc.)

The F2838xD family of devices contains 2 C28x CPU subsystems and one CM (Connectivity Manager) subsystem. Each C28x CPU subsystem includes

1. One C28x CPU
2. One CLA (Control Law Accelerator) co-processor
3. One DMA (Direct Memory Access Unit)

Note that C28x1 and C28x2 cores are also referred to as CPU1 and CPU2 respectively. An important part of multicore debugging is deciding which CPU has ownership of which hardware peripherals. The F2838xD CPU includes multiple memory blocks, peripherals and GPIOs. Some of these can only be owned by a certain core, and some of these are allowed to be shared among multiple cores. All the resources that come under the "can be shared by multiple cores" category need to be assigned an explicit owner before performing any function.

3 General Flow for a dual-core application

- An important rule of multi-core debugging is that this ownership is always decided by CPU1.
- Another important aspect is to enable effective communication and data sharing between the 2 CPUs. This is taken care of by the Inter Processor Communication Module. In addition to this, the device has additional dedicated message RAMs which can be used to share data.
- Even though both the cores have identical hardware - they have different levels of autonomy when it comes to the multi-core system. CPU1 acts as the master core. The basic initialization of the device, including the clocking and the initialization of all the GPIOs, is done by CPU1's application. CPU1 can be booted in different modes based on the boot pin configuration. The CPU2 and CM (in case of the F2838Xd) cores must be booted by CPU1 application using the IPC module.

4 Boot Sequence

When the device comes out of reset, only CPU1's boot ROM executes. CPU2 is held in reset by hardware and cannot run any code until CPU1 explicitly releases it. This means:

- **System clocks and PLL** are initialized exclusively by CPU1. Until CPU1 configures the PLL and waits for it to lock, the device is running on the internal oscillator at reduced speed. CPU2 inherits whatever clock state CPU1 has established.
- **Peripheral ownership** is assigned by CPU1. The device has a set of peripherals that can be assigned to either core - but the assignment registers are write-protected from CPU2. Only CPU1 can decide which core owns which peripheral.
- **GPIO initialization** is done by CPU1. All GPIO mux, direction, and pull-up settings must be configured by CPU1 before CPU2 can safely use any GPIO.
- **Watchdog** for the overall system is managed by CPU1. CPU2 has its own watchdog, but system-level protection starts with CPU1.

On every reset, the following sequence occurs:

1. **CPU1 boot ROM executes.** The boot ROM reads the boot mode pins (GPIO84, GPIO72, etc. depending on the device) and determines how to boot - JTAG, SCI, SPI, CAN, I2C, parallel, or directly from flash. This is the bootloader phase and it runs entirely on CPU1.
2. **CPU1 application starts.** Once the bootloader hands off, CPU1 begins executing the application. The first thing a well-written CPU1 application does is:
   - – Initialize the device (PLL, clocks, flash wait states)
   – Assign peripheral and memory ownership
   – Initialize any shared memory regions that CPU2 will need
3. **CPU1 releases CPU2 from reset.** CPU2 cannot start on its own. CPU1 must call the boot sequence for CPU2 - in driverlib this is Device_bootCPU2(). This function writes to the IPC registers to signal CPU2's boot ROM to execute and specifies the boot mode for CPU2 (typically boot from flash or boot from a specific RAM address loaded by CPU1).
4. **CPU2 boot ROM executes.** CPU2's boot ROM runs its own boot sequence, using the boot mode that CPU1 specified via IPC. If CPU1 directed CPU2 to boot from flash, CPU2 jumps to its flash entry point. If CPU1 directed CPU2 to boot from SARAM (useful in RAM debug builds where CPU1 has already copied CPU2's image), CPU2 jumps to the loaded code.
5. **Both cores run independently.** After this point, both CPUs execute their respective applications concurrently. Coordination between the two is handled entirely through IPC flags and shared/message RAM - there is no further hardware dependency between them during normal execution.

5 Running a simple dual core example

A good place to start understanding the concepts related to the dual-core environment would be to run a simple led blinky example on a dual core c28x device. For this example, we will be using the F28P65x launchpad. You will need a recent version of c2000ware and Code Composer Studio to be able to run this example.

How to blink an LED with CPU1 and CPU2?

Step 1 - Import Both Projects

Navigate to C:/ti/C2000Ware_x_x_x_x/driverlib/f28p65x/examples/c28x_dual/led/CCS/, and click on select folder. 2 different projects should appear. Make sure you import **led_ex1_c28x_dual_blinky.projectspec**. This will import both the cpu1 and cpu2 projects into your CCS workspace.

Step 2 - Choose your build configuration

To Debug in RAM Configuration

1. Change both project's build configuration to CPUx_LAUNCHXL_RAM
2. Build CPU1 project first, then CPU2. Both should build clean, and produce led_ex1_c28x_dual_blinky_cpu1\CPU1_LAUNCHXL_RAM\led_ex1_c28x_dual_blinky_cpu1.out and led_ex1_c28x_dual_blinky_cpu2\CPU2_LAUNCHXL_RAM\led_ex1_c28x_dual_blinky_cpu2.out respectively.
3. Then right-click on the target configuration file (.ccxml file) and click on start a project-less debug.
4. Once you have done this, it will start a CCS debug session, but no program will be loaded on any of the cores. You should see all the available cores in "Threads" section of the debug view.
5. Connect C28xx_CPU1 and C28xx_CPU2 (in that order) by right clicking on them in the "THREADS" section and choosing the "Connect Target" option.
6. Once both the cores are connected, select C28xx_CPU1 and go to Run → Load → Load Program → Navigate to and choose led_ex1_c28x_dual_blinky_cpu1.out
7. Once you do this, you'll see that the CPU1 program execution is halted at main(). Let the program run and you should see LED4 on your launchpad blinking.
8. It is very important that the CPU1 code runs past the Device_bootCPU2() function before running CPU2 code.
9. You can keep the CPU1 code running (or halt it at a point after the Device_bootCPU2() function), and then load led_ex1_c28x_dual_blinky_cpu2.out image to C28xx_CPU2.
10. If all the steps were followed correctly, it should halt at main() as well and then you should be able to run it normally. You will see LED5 on your launchpad blinking.

To Debug in Flash Configuration

1. Change both project's build configuration to CPUx_LAUNCHXL_FLASH
2. Start a project-less debug in CPU1 target configuration .ccxml file.
3. Connect target on CPU1 and CPU2 threads.
4. Go to properties of CPU1 and open flash bank flash settings in the dropdown. Scroll down to Flash Bank Map Settings and set 1 for banks 3 and 4 (0 is for CPU1 and 1 is for CPU2).
5. Click on configure clock.
6. Scroll down to erase settings and check on Selected banks only -> then select banks 0,1,2 (they are meant for CPU1).
7. Now click on CPU2 properties. If Flash Bank Map Settings is enabled, once again set 1 for bank 3 and 4. Else if it is disabled, leave it as it is.
8. Again, scroll down to erase settings and select banks 3 and 4.If the debug console shows 'Device is in reset', unplug and plug the USB again, or open a new session. If the error still pops, the program will run irrespective.
9. Now click on CPU1 thread and go to Run -> Load -> Load program.
10. Now browse the respective .out file (from any previous build) and run the program to execute it. (NOTE Make sure that CPU1 program runs at least till function Device_bootCPU2(). Follow the same with CPU2 thread as well.

Common Issues faced while running the example

hex2000 utility unrecognized?

Sometimes it must be referenced using its original project path, specifically when using Windows powershell. Instead of just mentioning hex2000, provide the full path for hex2000.

After loading the combined.hex to cpu1, memory not updated in the correct address?

Boot ROM sequences differently at times, and CPU2 might still be at default boot address, check the Disassembly view and manually change the pointer location to check what has been filled in memory. In case memory is updated, disconnect the launchpad (triggers a reset) and connect again to check. If it has not been updated in memory, check the combined.map file to see if the combined output file has been built correctly. If not, import the project fresh and try again.

Both CPUs run but led is not blinking?

Set breakpoints to find out where the program is stopping. If it stops at IPCsync, rebuild the project and restart a project-less debug and load the program in the right order after connecting target of CPU2, and also make sure that the parameters passed to the IPCsync function of both CPUs are the same. Otherwise, also make sure that the sysconfig file in the CPU1 project folder has CPU2 boot mode set to Boot from Flash Bank 3, then save the file.

6 Key Pitfalls

1. **Boot sequence ownership:**

   CPU1 is always the master core and must boot first. CPU1 controls the reset of CPU2 and is responsible for all system initialization (PLL, clocks, watchdog). You cannot just connect CCS to CPU2 cold - CPU1 must be running and have released CPU2 from reset.
2. **Load order in CCS matters:**

   Connect and load CPU1 first, run it far enough to initialize the system and boot CPU2 (or at minimum release CPU2 from reset), then connect and load CPU2. If you load CPU2 before CPU1 has initialized the system clocks, you will get erratic behavior.
3. **Separate CCS projects:**

   A dual-core application is two completely independent images (out files). They are linked only through shared memory and IPC - there is no single "dual-core project." (unless you use the post-build step in section 5)
4. **Memory ownership / access rights:**

   Shared RAM blocks have ownership assignments - if CPU1 owns a GS RAM block, CPU2 will get a memory access violation trying to write to it. The linker command files (.cmd) for CPU1 and CPU2 must be carefully coordinated so they don't overlap, and shared sections must use dedicated IPC/MSG RAM regions or properly owned GS RAM.
5. **Halting one core affects the other:**

   When you halt CPU1 at a breakpoint, CPU2 keeps running unless you explicitly halt it too (or use a Sync Group in CCS). IPC flag polling loops on the running core may spin indefinitely if the halted core is the one expected to respond. This is the single most common source of confusing behavior during dual-core debug.
6. **Global breakpoints / Sync Groups:**

   CCS supports grouping cores so a breakpoint on one halts both simultaneously. For IPC-heavy code, this is almost always what you want - otherwise the running core races ahead, consumes IPC messages, and state becomes inconsistent by the time you look at it.
7. **Watchdog:**

   Each core has its own watchdog. If you halt CPU1 for debugging but CPU2 is still running and servicing a shared peripheral, or vice versa, the unwatched core's watchdog may fire and reset the device. Disable watchdogs early in debug builds.
8. **Flash vs. RAM:**

   In flash builds, both cores share the flash controller, and concurrent flash reads from both cores at the same pipeline stage can cause stalls. During debug, most teams work from RAM to avoid this and to enable fast re-load without erase cycles. Flash programming requires both cores to be coordinated (CPU1 typically does all flash init).
9. **IPC race conditions:**

   IPC flag-based protocols are asynchronous by nature. A common bug: CPU1 sets a flag and then immediately writes data to a shared buffer, but CPU2 reads the buffer before CPU1 finishes writing. Use IPC flags only to signal after the data write is complete and use memory barriers (__asm(" RPT #7 || NOP") on C28x) where needed.
10. **CLA is a third debuggable core:**

    On devices where a CLA is present, it appears as a separate debug target in CCS. Load CLA symbols from the CPU2 .out (they're compiled together). CLA has no RTOS, no stack, and very limited debug visibility - you can step and see registers but there's no call stack.
11. **Heterogeneous debugging (F2838x CM core):**

The ARM CM core uses a completely different tool chain. In CCS, it appears as a separate ARM Cortex-M4 target. You need an ARM compiler project alongside your C28x projects. The CM core cannot directly access C28x local RAMs - all data exchange goes through MSG RAMs.

7 One-click solution for dual core flashing

This section explores how we can combine two binaries (one for CPU1 and one for CPU2) and load them directly to the desired regions in Flash, in just one step. We will just load the combined binary to CPU1 as opposed to manually loading each of the binary to its respective core. This is achieved using a post-build step.

To generate a combined output file to load on one CPU and control the other CPU directly in F28P65x (For users who haven't downloaded the latest c2000ware version)

- Import the led_ex2_blinky_sysconfig_multi from the C2000 f28p65x, which automatically imports both cpu1 and cpu2 projects into your workspace.
- Right-click on the multi folder and select the desired Build Configuration.
- Open the **.sysconfig** file in the cpu1 project folder and set CPU2 boot mode to Boot from Flash Bank 3, then save the file.
- Right-click the **sysconfig_multi** project and select Build Projects.
- Now, manually enter the following command as a postBuildStep under right-click CPU2 → Properties → Build → Steps → post-build step and save it. The command is:

  ${CG_TOOL_ROOT}/bin/hex2000.exe --memwidth=16 --romwidth=16 -ii --map="..\..\combined.map" "..\..\led_ex2_blinky_sysconfig_cpu1\"FLASH_LAUNCHXL or FLASH"\led_ex2_blinky_sysconfig_cpu1.out" "..\..\led_ex2_blinky_sysconfig_cpu2\"FLASH_LAUNCHXL or FLASH"\led_ex2_blinky_sysconfig_cpu2.out" -o "..\..\combined.hex"
- Open the target configuration file in the cpu1 **targetConfigs** folder and click Start Project-less Debug."
- Now right-click on CPU1 and CPU2 to connect to the target.
- Right-click on CPU1 -> Flash settings -> Scroll down to Flash Bank Map Settings -> Make sure all are set to 0 (we are loading the entire data from CPU1 itself, so it needs access to all banks)
- Scroll down further to check select Entire flash under Erase settings.
- Scroll up to click Configure Clock and then, Save and Close.
- Now click on Run -> Load program -> Browse -> combined.hex
- Open View → Memory Browser and verify that data has been loaded correctly at 0x8000 for CPU1 and 0xE0000 for CPU2.
- Click Run on both CPU1 and CPU2 to start execution, then verify the output using Launchpad LEDs or the Output Console.

If user has downloaded the latest c2000ware version (Will be available by the next release)

Directly building the multi project will create the combined.hex and combined.map file inside the multi project folder, for running the example, the above project-less debug can be followed.

Merging the postBuildStep in the projectspec:

The following change was made to the

"examples_driverlib\c28x_dual\led\sysconfig_led_ex_f28p65x\CCS\led_ex2_blinky_sysconfig.projectspec",

which is in the c2000-device-support repository, for merging into the SDK.

For verification on the launchpad, the same change has to be made in the

"C:\ti\c2000\C2000Ware_26_01_00_00\driverlib\f28p65x\examples\c28x_dual\led\CCS\led_ex2_blinky_sysconfig.projectspec" file, which is inside the c2000ware folder.

- <configuration name="FLASH" compilerBuildOptions="--opt_level=off -I${C2000WARE_ROOT} -I${PROJECT_ROOT}/device -I${C2000WARE_DLIB_ROOT} --define=_FLASH -v28 -ml -mt --cla_support=cla2 --float_support=fpu64 --tmu_support=tmu1 --define=DEBUG --define=CPU2 --gen_func_subsections=on --diag_warning=225 --diag_suppress=10063" linkerBuildOptions="--entry_point code_start --stack_size=0x3F8 --heap_size=0x200 --define=_FLASH "

  postBuildStep="${CG_TOOL_ROOT}/bin/hex2000.exe --memwidth=16 --romwidth=16 -ii --map='..\..\led_ex2_blinky_sysconfig_multi\combined.map' '..\..\led_ex2_blinky_sysconfig_cpu1\FLASH\led_ex2_blinky_sysconfig_cpu1.out'

'..\..\led_ex2_blinky_sysconfig_cpu2\FLASH\led_ex2_blinky_sysconfig_cpu2.out' -o '..\..\led_ex2_blinky_sysconfig_multi\combined.hex'"

/>

<configuration name="FLASH_LAUNCHXL" compilerBuildOptions="--opt_level=off -I${C2000WARE_ROOT} -I${PROJECT_ROOT}/device -I${C2000WARE_DLIB_ROOT} --define=_FLASH -v28 -ml -mt --cla_support=cla2 --float_support=fpu64 --tmu_support=tmu1 --define=DEBUG --define=CPU2 --gen_func_subsections=on --diag_warning=225 --diag_suppress=10063" linkerBuildOptions="--entry_point code_start --stack_size=0x3F8 --heap_size=0x200 --define=_FLASH "

postBuildStep="${CG_TOOL_ROOT}/bin/hex2000.exe --memwidth=16 --romwidth=16 -ii --map='..\..\led_ex2_blinky_sysconfig_multi\combined.map' '..\..\led_ex2_blinky_sysconfig_cpu1\FLASH_LAUNCHXL\led_ex2_blinky_sysconfig_cpu1.out' '..\..\led_ex2_blinky_sysconfig_cpu2\FLASH_LAUNCHXL\led_ex2_blinky_sysconfig_cpu2.out' -o '..\..\led_ex2_blinky_sysconfig_multi\combined.hex'"

/>

The above change has been made as a post-build step to cpu2 project.

How is a single output file loading in f28p65?

- hex2000 merges the two CPU output files and automatically strips the EOF record from the first file, so the second file can follow seamlessly in a single Intel-HEX image.
- The resulting HEX contains CPU 1's code at 0x08000 (Flash Bank 0) and CPU 2's code at 0xE0000 (Flash Bank 3); one flash operation programs both banks together.
- After loading, the start vectors are re-pointed: CPU 1 begins execution from its flash address, then releases the boot for CPU 2, which boots from Flash Bank 3.
- CPU 1 is granted write access to all five flash banks, giving it full control to program its own region and the region used by CPU 2.
- Loading is done with a single **Run → Load Program** command; after the load you can open the Memory Browser to confirm data at 0x08000 for CPU 1 and 0xE0000 for CPU 2.
- When execution starts, CPU 1 halts, triggers CPU 2's boot, and only proceeds once CPU 2 is running, guaranteeing coordinated dual-CPU operation.

Why will this combined hex approach not work for other dual core devices?

The same approach to merge CPU1 and CPU2 binaries may not work due to different memory ownership. For example, take the case of the F2838x device

f2838x flash memory:

CPU	Physical/global flash address	Size	Local address
CPU1	0x200000	512KB	0x080000 (same address which CPU2 also sees)
CPU2	0x300000	512KB	0x080000 (same address which CPU1 also sees)

- Here both CPUs have their own *local* memory map, each CPU sees its own flash start at **0x080000**. That's why a linker map generated from a single-CPU build shows both CPUs appear to start at the same address.
- **hex2000** merges files based on the *local* addresses that appear in the .out files. Since both CPUs report 0x08000 as the start of flash, merging without address remapping causes one image to overwrite the other.
- To program them correctly, CPU2 sections must be remapped to global base before merging.

We are currently working on implementing a similar solution for other dual core devices.

8 Inter Processor Communication

IPC is the hardware mechanism by which the cores signal and pass data to each other. Key components of the IPC are: -

- **IPC Flag Registers** - 32 hardware flag bits per core pair (CPU1↔CPU2, CPU1↔CM, CPU2↔CM). One core sets a flag; the other core gets an interrupt or can poll the flag. This is used for lightweight signaling ("data is ready", "command received").
- **Message RAMs (MSG RAMs)** – This is a dedicated SRAM physically accessible by both cores, and is used to pass data. Each direction has its own MSG RAM (CPU1-to-CPU2 MSG RAM is write-protected from CPU2's side to avoid corruption).
- **Global Shared RAM (GS RAM)** – These are larger shared memory blocks and are accessible by multiple cores. Ownership for this memory can be assigned dynamically. Not available between C28x and CM cores (only MSG RAMs are used there).
- **IPC Driver / driverlib** - C2000Ware provides a software IPC driver that layers a message-passing protocol (command + address + data) on top of the hardware flags.

More information related to each device's specific IPC registers can be found in the device TRM.

9 Helpful Links

General Debugging tips for C28x devices - C2000 Debugging Tips & Tricks

Debugging Multiple cores - 6. Debugging multiple cores — C2000™ Multicore Development Guide

Relevant FAQs

(4) TMS320F28388D: using debugger with 2 cores - C2000 microcontrollers forum - C2000™ microcontrollers - TI E2E support forums

(4) TMS320F28384S: Debugging multiple cores problem - C2000 microcontrollers forum - C2000™ microcontrollers - TI E2E support forums

(4) TMS320C28345: C28p65 Dual core CPU2 Debug Issue - C2000 microcontrollers forum - C2000™ microcontrollers - TI E2E support forums

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, 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 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, TI’s General Quality Guidelines, 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. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

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

Last updated 10/2025