

Chapter 6: Dual Core

After completing this chapter, you will understand how to create dual core applications for PSoC™ 6 devices.

Note: There are no dual core versions of the PSoC™ 4, so in this chapter we will only discuss PSoC™ 6.

Table of contents

| | | |
|--|--|----------|
| 6.1 | PSoC™ 6 Processors | 2 |
| 6.1.1 | Cortex®-M0+ | 2 |
| 6.1.2 | Cortex®-M4F | 2 |
| 6.1.3 | Memory | 2 |
| 6.2 | Application directory structure | 3 |
| 6.3 | Inter-Process Communication (IPC) | 5 |
| 6.3.1 | Semaphores | 5 |
| 6.3.2 | Pipes | 6 |
| 6.4 | Debugging | 7 |
| 6.5 | Exercises | 8 |
| Exercise 1: Run the semaphore code example | | 8 |
| Exercise 2: Run the pipes code example | | 9 |

Document conventions

| Convention | Usage | Example |
|-------------------|--|--|
| Courier New | Displays code and text commands | <code>CY_ISR_PROTO(MyISR);</code> <code>make build</code> |
| <i>Italics</i> | Displays file names and paths | <i>sourcefile.hex</i> |
| [bracketed, bold] | Displays keyboard commands in procedures | [Enter] or [Ctrl] [C] |
| Menu > Selection | Represents menu paths | File > New Project > Clone |
| Bold | Displays GUI commands, menu paths and selections, and icon names in procedures | Click the Debugger icon, and then click Next . |

6.1 PSoC™ 6 Processors

PSoC™ 6 devices are equipped with two processors, an ARM® Cortex®-M0+ (CM0+) and an ARM® Cortex®-M4F (CM4). Together these cores allow you to simultaneously optimize your device for power and performance. Whether you need an ultra-low power programmable solution or a high-performance, secure, dual-CPU solution, the PSoC™ 6 family can support your needs.

By default, when creating a single core application for PSoC™ 6 devices, the code you write will be executing on the CM4 core. If you want to create a single core application that uses the CM0+ core, you need to create a dual core application and then simply not enable the CM4 core.

The following document contains a description of the PSoC™ 6 dual core architecture, as well as the dual core application structure: <https://www.cypress.com/file/385711/download>

6.1.1 Cortex®-M0+

The CM0+ is a 32-bit processor whose design is focused on minimal power consumption. This 100 MHz processor enables single-cycle integer multiplication and comes equipped with a memory protection unit (MPU). For more information on this core, you can refer to:

<https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m0-plus>

In PSoC™ 6 devices, this is always the first core to boot up. In single core applications, it simply starts the CM4 and then goes to sleep, but in dual core applications, it remains awake and will execute whatever code you have written for it.

6.1.2 Cortex®-M4F

The CM4 is a 32-bit processor with high performance signal processing capabilities. This 150 MHz processor enables single-cycle floating point multiplication and comes equipped with a digital signal processing unit (DSP) and an MPU. For more information on this core, you can refer to:

<https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m4>

6.1.3 Memory

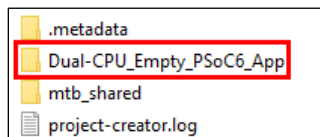
The amount of RAM and flash memory that is allocated to each CPU, as well as their stack and heap sizes, is configurable. The CAT1 PDL documentation contains a detailed guide on how to configure these values:

https://infineon.github.io/mtb-pdl-cat1/pdl_api_reference_manual/html/group_group_system_config.html

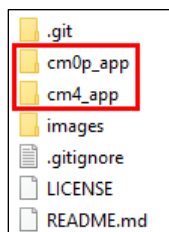
Shared memory can be allocated simply by declaring a global variable on one CPU, then passing the address of that variable to the other CPU using inter-processor communication (IPC).

6.2 Application directory structure

The directory structure of a dual core application is different from that of a single core application. If you create a new dual core application, then navigate to your ModusToolbox™ workspace directory, you will see a top-level directory like this:



This is the application directory for the dual core application. If you navigate the top-level directory, you'll see something like this:

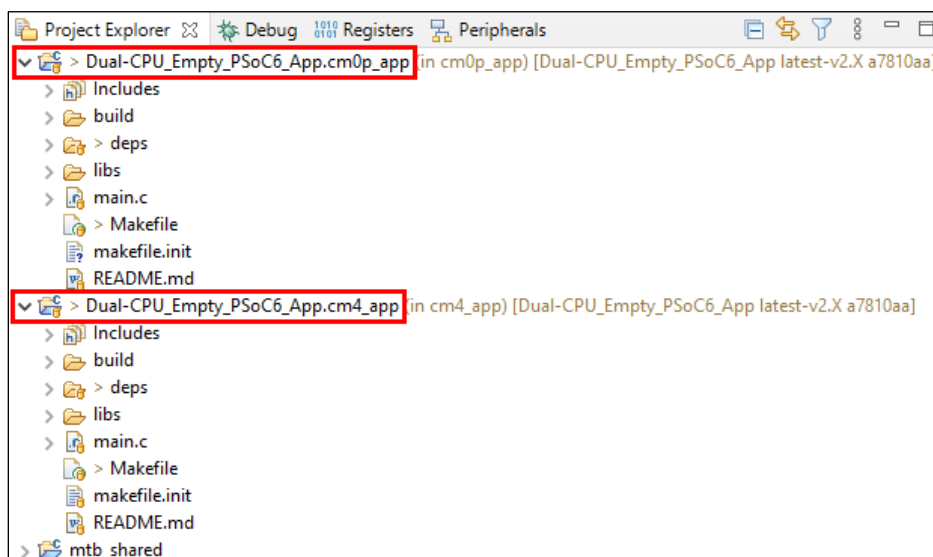


Within the application directory, there are two project directories, one for each of the cores in the PSoC™ 6 device.

When compiling a dual core application, two sets of build output files are created, one for the CM0+ and one for the CM4. Each built output set is compiled from its respective project within the dual core application.

Note: Single core PSoC™ 6 applications technically have two executables as well, but in single core applications the CM0+ executable simply wakes up the CM4 and then puts the CM0+ to sleep.

Inside the Eclipse IDE for ModusToolbox™, the application described above would appear like this:



As you can see, there is a complete project for each core in the PSoC™ 6 Device, each core has its own *deps* directory, *Makefile*, and *main.c*. However, both of these projects belong to the same dual core application.

Note: *In the example above, there is a REAME.md at the application level and one inside each individual project. The ones inside the project can be seen from inside the Eclipse IDE for ModusToolbox™, but the one at the application level will not appear. If you want to look at the application level README.md file, you must open it from the file system using a markdown viewer.*

Including Code

Libraries can be added independently to each core's project via the Library Manager, so code that is included for one processor does not have to be included in the others.

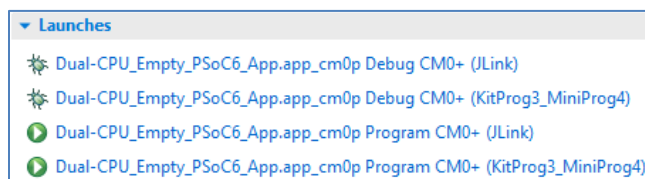
If you have code that you want to include for both processors, you should place it in the application directory above the individual project directories, and then `#include` it in each project.

BSP and Device Configuration

Both the CM0+ and the CM4 projects contain a BSP, however, only the BSP in the CM4 project is used. So, when editing the device configuration for a dual core application, you should edit the *design.modus* file present in the CM4 project. That is, you should select the CM4 project before launching the Device Configurator. If you create a custom device configuration, you only need to set the *Makefile* `COMPONENTS` and `DISABLE_COMPONENTS` variables in the CM4 project's *Makefile*.

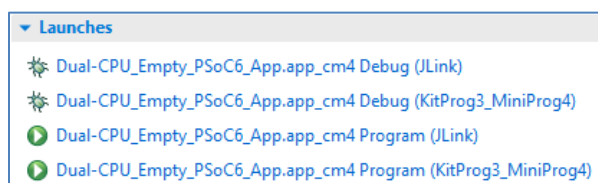
Launch Configurations

Each of a dual core application's projects comes with its own set of launch configurations. The following are the launch configurations for the CM0+ project:



These launch configurations allow you to program and debug the CM0+, but not the CM4.

The following are the launch configurations for the CM4 project:



These launch configurations will program both the CM0+ and the CM4 at once, but they only allow you to debug the CM4.

Note: *With regard to device and launch configurations, the CM4 project can be thought of as the applications "primary" project.*

6.3 Inter-Process Communication (IPC)

IPC is the method by which CPUs communicate with each other. On PSoC™ 6, there are two IPC methods which you can make use of in your applications:

- Semaphores
- Pipes

We will discuss these in more detail later in this section.

On PSoC™ 6 devices, IPC is implemented in hardware. The CAT1 PDL documentation contains an in-depth look at how IPC is implemented, what you need to consider when configuring it, and how to configure it:

https://infineon.github.io/mtb-pdl-cat1/pdl_api_reference_manual/html/group_group_ipc.html

Rather than repeat all of that information here, we will simply discuss the basics that you need to know to get your application up and running.

6.3.1 Semaphores

A semaphore is a signaling mechanism between threads. The name semaphore (originally used for sailing ship signal flags) was applied to computers by Dijkstra in a paper about synchronizing sequential processes. Semaphores can either be binary or counting. PSoC™ 6 IPC semaphores are binary semaphores. When you "set" a semaphore it attempts to acquire the semaphore but can only do so if someone else has not acquired it first. When you "clear" a semaphore it releases the semaphore.

When your PSoC™ 6 device boots up, the default system initialization routines create an array of 128 semaphores (four 32-bit values). The first 16 semaphores (semaphores 0-15), are reserved for system use. The rest of the semaphores however, are available for use by your application. To make use of these semaphores you need the following functions:

- `Cy_IPC_Sema_Set` – Acquire a semaphore
- `Cy_IPC_Sema_Clear` – Release a semaphore

These functions take the following arguments:

- `uint32_t semaNumber` – The index of the semaphore to acquire or release
- `bool preemptable` – When this parameter is enabled the function can be preempted by another task or other forms of context switching in an RTOS environment

It is important to note that these functions are non-blocking. When called, they immediately return one of the following values:

- `CY_IPC_SEMA_SUCCESS` – The semaphore was set successfully
 - `CY_IPC_SEMA_LOCKED` – The semaphore channel is busy or locked by another process
 - `CY_IPC_SEMA_NOT_ACQUIRED` – Semaphore was already set
 - `CY_IPC_SEMA_OUT_OF_RANGE` – The semaphore number is not valid
- `Cy_IPC_Sema_Status` – Gets the status of a semaphore

This function takes the following argument:

- `uint32_t semaNumber` – The index of the semaphore to query

This function returns one of the following values:

- `CY_IPC_SEMA_STATUS_LOCKED` – The semaphore is in the set state.
- `CY_IPC_SEMA_STATUS_UNLOCKED` – The semaphore is in the cleared state.
- `CY_IPC_SEMA_OUT_OF_RANGE` – The semaphore number is not valid

6.3.2 Pipes

A pipe is a communication channel that allows you to transfer messages or data between the cores in the PSoC™ 6. Pipes can transfer a single 32-bit unsigned datum, an array of data, or even user-defined structures. Pipes can be configured in full-duplex mode so that both cores can send messages to each other through the same pipe, or in single direction mode, so that one core will be the sender and the other will be the receiver.

Each pipe is configured with exactly two "endpoints", one on each core. Whenever a message is sent through a pipe, it is really just being sent between endpoints. Each endpoint then has one or more "clients" associated with it. Clients merely consist of a client ID and an associated callback function. When messages are sent through a pipe, they are addressed to specific clients. In this way, your application can define a multitude of message types that each have their own unique callback function that will be run when that message is received. (Each endpoint has an array of client callback functions; the client ID is simply an index into this array specifying which callback to run). There is no limit to the number of clients that an endpoint can have.

The first piece of data in any message sent through a pipe is required to be the client ID, specified as a 32-bit unsigned integer. This client ID specifies the callback that should be executed by the receiving endpoint. Optionally, after the client ID, you can include as much data as you want in the message.

When an endpoint receives a message, that endpoint's message received ISR is automatically run. This ISR is configurable, and can perform any function required by your application, but before it finishes, it must call the function `Cy_IPC_Pipe_ExecuteCallback`. This function retrieves the client ID included in the received message and runs the associated callback function. (Again, the client ID is simply an index into an array of callback functions).

After the receiving endpoint's client callback has been run, another callback, called the "release callback", will be automatically run. The release callback is defined by the sender of the message and runs on the sender's CPU.

The following is a high-level list of the steps you need to take to set up a pipe and send a message:


1. Call the function `Cy_IPC_Pipe_Init` to configure the endpoints of the pipe. This function must be called by both CPUs.
2. Call the function `Cy_IPC_Pipe_RegisterCallback` to register your endpoint's clients. You will need to call this function once for every client you wish to register. This must be done on both CPUs.
3. Optionally, call the function `Cy_IPC_Pipe_RegisterCallbackRel` to register your endpoint's release callback. The release callback can also be specified as an argument of the `Cy_IPC_Pipe_SendMessage` function, so this step is optional.
4. Call the function `Cy_IPC_Pipe_SendMessage` to send a message through the pipe. You don't have to make any API calls to receive the message; the receiving endpoint's message received ISR will automatically be run when the message is received.

The CAT1 PDL contains a detailed description, as well as a plethora of code snippets, showing how to create, configure, and use your own custom pipes. Infineon also provides a code example that shows the complete process of creating, configuring, and using pipes. We will look at this CE in a later exercise.

6.4 Debugging


Debugging code running on two cores at the same time may be a complex process. A device such as an oscilloscope or a logic analyzer may be useful for monitoring communication between the CPUs. Currently, the Eclipse IDE for ModusToolbox™ only allows you to debug one CPU at a time. When debugging a dual core application, it is recommended that you first debug the portions of the code where the CPUs communicate with each other; after that, code executed by individual CPUs can be debugged separately.

To debug the CM4, select the CM4 project you want to debug in your workspace, then simply run the Debug launch configuration from the Quick Panel:


 [Dual-CPU_Empty_PSoC6_App.cm4_app Debug \(KitProg3_MiniProg4\)](#)

This launch configuration will program both the CM0+ and the CM4 and then start debugging the CM4.

To debug the CM0+, the first thing you need to do is program the complimentary CM4 application to your board. To do this, in your workspace, select the CM4 project associated with the CM0+ project you want to debug. Then, run the Program launch configuration from the Quick Panel:

 [Dual-CPU_Empty_PSoC6_App.cm4_app Program \(KitProg3_MiniProg4\)](#)

This will program both the CM0+ and the CM4. Then, in your workspace, select the CM0+ project you want to debug and run the Debug configuration from the Quick Panel:

 [Dual-CPU_Empty_PSoC6_App.cm0p_app Debug CM0+ \(KitProg3_MiniProg4\)](#)

This launch configuration will reprogram the CM0+ if necessary, and then begin debugging the CM0+.

6.5 Exercises

Exercise 1: Run the semaphore code example

In this exercise we will examine and run the Infineon PSoC™ 6 semaphore code example. This CE waits for user button 1 to be pressed. Once the button is pressed each processor attempts to acquire a semaphore which locks access to the debug UART. When the semaphore is acquired by a processor, that processor sends a message over the UART, then releases the semaphore.

- ☐ 1. Create a new application called **ch06_ex01_semaphore** using the code example **Dual-CPU IPC Semaphore**.
- ☐ 2. Open *main.c* from both of your application's projects and read through the code. Make sure that you understand what is going on.

Note: Remember that the CM0+ is the first processor to boot up.

- ☐ 3. At the top of each *main.c* file is a line that says: `#include "ipc_def.h"`
Right click on the file name and select **Open Declaration**. Take a look at what is in this file.
- ☐ 4. Once you understand how the project works, program the project to your kit and verify that when you press the user button, you see messages from both cores.

Note: To program both cores, select the CM4 project, then select the Program launch configuration that doesn't end in "CM0+".

Questions to Answer

- ☐ 1. What is the purpose of the call to the function `Cy_SysEnableCM4` in the CM0+'s code?
- ☐ 2. Why do both of the processors need to call the function `__enable_irq`?
- ☐ 3. Where is the file *ipc_def.h* saved on your disk?

Exercise 2: Run the pipes code example

In this exercise we will examine and run the Infineon PSoC™ 6 pipes code example. This CE sets up an IPC pipe, then waits for user button 1 to be pressed. Once the button is pressed the CM4 sends a message to the CM0+ to start generating random numbers. The CM0+ then generates random numbers and sends messages to the CM4 containing the random numbers. The CM4 then prints each number over the Debug UART. This continues until the user button is pressed again, at which point the CM4 sends a message to the CM0+ to stop generating random numbers.

- ☐ 1. Create a new application called **ch06_ex02_pipes** using the code example **Dual-CPU IPC Pipes**.
- ☐ 2. Open *main.c* from both of your application's projects and read through the code. Open the CAT1 PDL and look up any functions you are unfamiliar with. Make sure that you understand what is going on.

Note: Remember that the CM0+ is the first processor to boot up.

- ☐ 3. At the beginning of each *main.c* file there is a function call to set up the IPC communication for the core.
 - o `setup_ipc_communication_cm0`
 - o `setup_ipc_communication_cm4`

These functions are defined in:

`<ApplicationDirectory>/shared/source/COMPONENT_CM<X>/ipc_communication_cm<X>.c`

Where <X> is the specific core, either "CM0P" or "CM4".

Open these files and read through the pipe initialization routine. Open the CAT1 PDL and look up any functions you are unfamiliar with. Make sure you understand what is going on here.

- ☐ 4. Once you understand how the project works, program the project to your kit and verify that when you press the user button, you see messages from both cores.

Note: To program both cores, select the CM4 project, then select the Program launch configuration that doesn't end in "CM0+".

Questions to Answer

- ☐ 1. In the file `<ApplicationDirectory>/shared/source/COMPONENT_CM0P/ipc_communication_cm0p.c`, what is the purpose of the function `user_ipc_pipe_isr_cm0`? When does this function get called?
- ☐ 2. The macros `USER_IPC_PIPE_EP_ADDR_CM0` and `USER_IPC_PIPE_EP_ADDR_CM4` correspond to which endpoints in the array of endpoint structures?
- ☐ 3. When the CM4 receives a message from the CM0+, what callback(s) are run?

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Published by
Infineon Technologies AG
81726 Munich, Germany

© 2021 Infineon Technologies AG.
All Rights Reserved.

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.