Chapter 7: 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, and one core on the CYW20829 is reserved for Bluetooth® so in this chapter we will only discuss PSoC™ 6.*

Table of contents

Document conventions

Convention	Usage	Example
`Courier New`	Displays code and text commands	`CY_ISR_PROTO(MyISR);` `make build`
Italics	Displays file names and paths	*sourcefile.hex*
[**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**.

7.1 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: PSoC™ 6 MCU Dual-CPU System Design

7.1.1 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.

7.1.2 Cortex®-M4

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

7.1.3 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).

7.2 Command Line Operations

The `-j` option should not be used on command line program operations on dual-core applications because it can lead to build and program operations occurring in the wrong order. That is, you should not use `make -j program` from the command line. Instead, you can use `-j` for the build followed by `qprogram`. For example: `make -j build; make qprogram`.

7.3 HAL usage

The hardware abstraction layer automatically reserves and configures resources as needed. For example, when the HAL is used to initialize a PWM, it reserves and configures one or more pins, a PWM block and a clock. The hardware manager keeps track of the resources that have been used so that they are not inadvertently re-used (potentially with an incompatible configuration) by other HAL initialization calls.

However, the hardware manager is only aware of resources allocated for the core that it is running on. Therefore, if you use the HAL on more than one core in an application, you must be careful that resource conflicts do not occur. There are two basic things you can do: (1) in some functions, rather than allowing the HAL to auto-assign resources such as clocks you can manually assign them; and (2) if the HAL uses a resource on one core, you can call the hardware manager on the other core(s) to reserve that resource so that the HAL knows they are already used elsewhere. See the HAL API Drivers section titled HWMGR for details on how to use the hardware manager.

7.4 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.

7.4.1 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. If enabled, the user must ensure that there are no deadlocks in the system, which can be caused by an interrupt that occurs after the IPC channel is locked. Unless the user is ready to handle IPC channel locks correctly at the application level, set to false.

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

7.4.2 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.

7.5 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. Use Project Creator to create a new application called **ch07_ex01_semaphore** using the code example **Dual-CPU IPC Semaphore** as the template.
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.

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. Use Project Creator to create a new application called **ch07_ex02_pipes** using the code example **Dual-CPU IPC Pipes** as the template.

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.
   - `setup_ipc_communication_cm0`
   - `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.

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?

7.6 Appendix

Answers to the questions asked in the exercises above are provided below.

7.6.1 Exercise 1 Answers

1) What is the purpose of the call to the function `Cy_SysEnableCM4`, in the CM0+'s code?

   This function call enables the CM4 CPU.

2) Why do both of the processors need to call the function `__enable_irq`?

   Because each CPU has its own NVIC.

3) Where is the file *ipc_def.h* saved on your disk?

   In *<ApplicationDirectory>/shared/include/ipc_def.h*

7.6.2 Exercise 2 Answers

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?

   This function gets called whenever a message is received by the CM0+ from the CM4. Its purpose is to call the callback function associated with the received client ID.

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?

   Endpoints 2 and 3.

3) When the CM4 receives a message from the CM0+, what callback(s) are run?

   The `user_ipc_pipe_isr_cm4` callback is run, then the `cm4_msg_callback` callback is 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

IMPORTANT NOTICE

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

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.