Peripheral Driver Library v3.0 User Guide

Doc. No. 002-18032 Rev. *J

Cypress Semiconductor

198 Champion Court

San Jose, CA 95134-1709

www.cypress.com

Contents

1. Introduction

Cypress provides the Peripheral Driver Library (PDL) v3.0 to simplify software development for the PSoC® 6 family of devices. The PDL is a complete software development kit that occupies the space between hardware and your application. It includes all required device-specific files, as well as higher-level middleware and RTOS support.

The core of the PDL is the source code that enables you to create custom drivers for peripherals on the device. To use a particular peripheral, add the driver source files to your project. You configure the peripheral to implement the desired functionality. You then use PDL API function calls to initialize, enable, and use the peripheral. The PDL also includes support for middleware such as Bluetooth Low Energy (BLE), and real-time operating systems (RTOS).

The PDL reduces the need to understand register usage and bit structures, thus easing software development for PSoC 6 MCU devices. If you develop code at the register level, you must understand how each peripheral uses which registers and pins, and the bit values required to control the peripheral correctly. You then write code to modify register values directly. The PDL makes this unnecessary.

1.1 Using this Guide

This guide provides a relatively high-level overview of the PDL. Details for each driver are in the PDL API Reference. See PDL Documentation for details about where to find the documentation, and how to use it.

Section 2, Peripheral Driver Library Overview, provides insight into the design of the PDL, the installed folder structure, and discusses the documentation provided.

Section 3, Build and Run a PDL Project, walks you through building a simple code example to ensure that everything is installed correctly and working properly.

There are two models for software development with the PDL, based on your choice of development environment.

- PSoC Creator™ Integrated Design Environment for both design and firmware
- PSoC Creator for the design, but another IDE to write firmware

Sections 4 and 5 provide a basic introduction to programming with the PDL, and cover the different considerations affecting each model.

Section 4, Developing Code Using the PDL, is useful for all developers without regard to the development environment.

Section 5, Using the PDL with PSoC Creator, discusses the considerations for using PSoC Creator to design your system and generate configuration code. This section is also of interest if you export code to another IDE to write the firmware. See Importing Generated Code into an IDE.

2. Peripheral Driver Library Overview

A PSoC 6 MCU device has an ARM® Cortex® M4 (CM4) and a Cortex M0+ (CM0+) processor in a single die, with multiple peripherals. In most cases, either core can use any peripheral. In some cases, there are multiple instances of a peripheral. For example, you can set up multiple serial communication blocks. And in some cases, any given instance may work on multiple sets of user data.

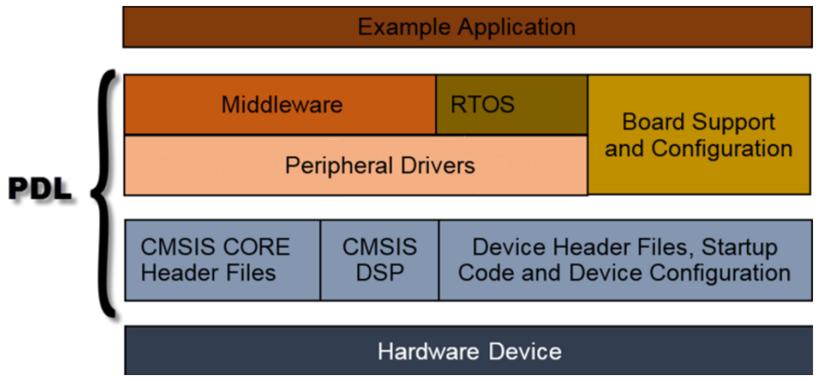
The PDL is designed to be both comprehensive and flexible to handle PSoC 6 MCU devices. The PDL provides the source code to customize any peripheral to implement the behavior you need, and ensure that it can work with whatever data is required. In addition, the PDL provides all device-specific files required for development.

The PDL delivers cryptographic functionality as a precompiled binary library, rather than as source code. In this case, header files provide the public API and the documentation describes how to use the cryptography engine.

The PDL also includes middleware and RTOS code that is fully integrated with the PDL. If you have your own middleware or preferred RTOS, use the resources provided as examples of how to integrate such code with the PDL.

Figure 1 is a block diagram of the main elements of the PDL.

Figure 1. PDL Design



Where to Get the PDL

PSoC Creator 4.2 installs the PDL. However, there may be a newer version of the PDL available. You can get the latest released version at the PDL product page.

2.1 PDL Peripheral Drivers

Table 1. Supported Peripherals

Peripheral	Description	API Functionality
CTB	Continuous Time Block	Configure and manage the analog continuous time block
CTDAC	Continuous Time Digital-to- Analog Converter	Configure and manage the 12-bit, continuous time digital to analog converter
CRYPTO	Cryptographic Accelerator	Perform cryptographic operations on user-designated data; PDL provides public header files and binary libraries
DMA	Direct Memory Access	Perform direct memory transfers
EFUSE	Electronic Fuses	Read the customer-accessible electronic fuses
FLASH	Flash Memory	Manage flash memory operations
GPIO	General Purpose I/O Ports	Configure and access device input/output pins
I2S	Inter-IC Sound	Manage digital audio streaming to external I2S devices
IPC	Inter-Processor Communication	Manage data transfer between CPUs or processes in a device
LPCOMP	Low-Power Comparator	Fast comparison of internal and external analog signals in all power modes
LVD	Low-Voltage Detect	Configure and manage low-voltage detection
MCWDT	Multi-Counter Watchdog Timer	Manage counters to create a free-running timer or periodic interrupts
PDM_PCM	PDM to PCM Converter	Convert one-bit digital audio streaming data to PCM data
PROFILE	Energy Profiler	Measure relative energy consumption of monitored operations
PROT	Memory Protection	Manage the MPU, Shared MPU (SMPU), and Peripheral Protection Unit (PPU)
RTC	Real Time Clock	Manage calendar date and clock time
SAR ADC	Successive Approximation Register Analog-to-Digital Converter	Configure and manage the 12-bit SAR ADC
SCB	Serial Communication Block	Manage serial communication as I^2C, SPI, or UART
SMIF	Serial Memory Interface	Manage an SPI-based interface to external memory devices
SYSANALOG	System Analog Reference	Generate highly-accurate reference voltages and currents for the analog subsystem
SYSCLK	System Clock	Manage system and peripheral clocks
SYSINT	System Interrupt	Manage interrupts and exceptions, in conjunction with the ARM Cortex Microcontroller Software Interface Standard (CMSIS) Nested Vectored Interrupt Controller (NVIC) API
SYSLIB	System Library	Utility functions to handle delays, register read/write, asserts, software reset, silicon unique ID, and more
SYSPM	System Power Management	Manage power modes and get power mode status
SYSTICK	ARM System Timer	Manage a 24-bit down-counter timer
TCPWM	Timer Counter PWM and Quadrature Decoder	Manage a 16- or 32-bit periodic counter, PWM, or Quadrature decoder
TRIGMUX	Trigger Multiplexer	Manage the multiplexing of trigger outputs to specific trigger inputs across multiple peripherals
WDT	Watchdog Timer	Manage the watchdog timer

A device-specific header file specifies the peripherals that are available for that device. If you write code that uses an unsupported peripheral, you will get a build-time error. Before writing code to use a peripheral, consult the device header file or datasheet for the particular series or device to confirm support for the peripheral.

2.2 PDL Organization

The PDL is organized into folders that closely follow the design of the PDL.

Figure 2. PDL Folders

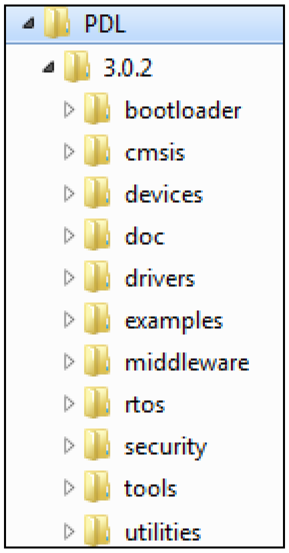


Table 2. PDL Folder Descriptions

Folder	Feature	Description
bootloader	Bootload support	Cypress bootloader SDK files
cmsis	CMSIS support	The ARM Cortex Microcontroller Software Interface Standard (CMSIS) core access header files, and DSP code (PDL 3.0 uses CMSIS v5.0)
devices	Device- and IDE-specific files	For each device package, common header files as well as configuration, startup, and project files for each supported IDE
doc	Documentation	PDL and middleware documentation
drivers	Driver source code	Source and header files for each PDL peripheral
examples	Code examples	Code examples organized by the supported starter kit
middleware	Software stacks	Firmware stacks, such as Bluetooth Low Energy (BLE) and emulated EEPROM
rtos	RTOS support	RTOS source code supported by the PDL, such as FreeRTOS
security	Secure system template	A basic secure system project template (PSoC Creator).
tools	Build support	User-level applications; for example to configure a software component or to perform post-build processing
utilities	Standard I/O	Various utility files such as standard I/O support

2.3 PDL Documentation

The installer puts the PDL documentation in the *doc* folder. In a default installation, the path is:

C:\Program Files (x86)\Cypress\PDL\3.0.x\doc\

You will find these PDL-related documents at the top level:

- pdl_user_guide.pdf – this document
- pdl_api_reference_manual.html
- pdl_release_notes.pdf

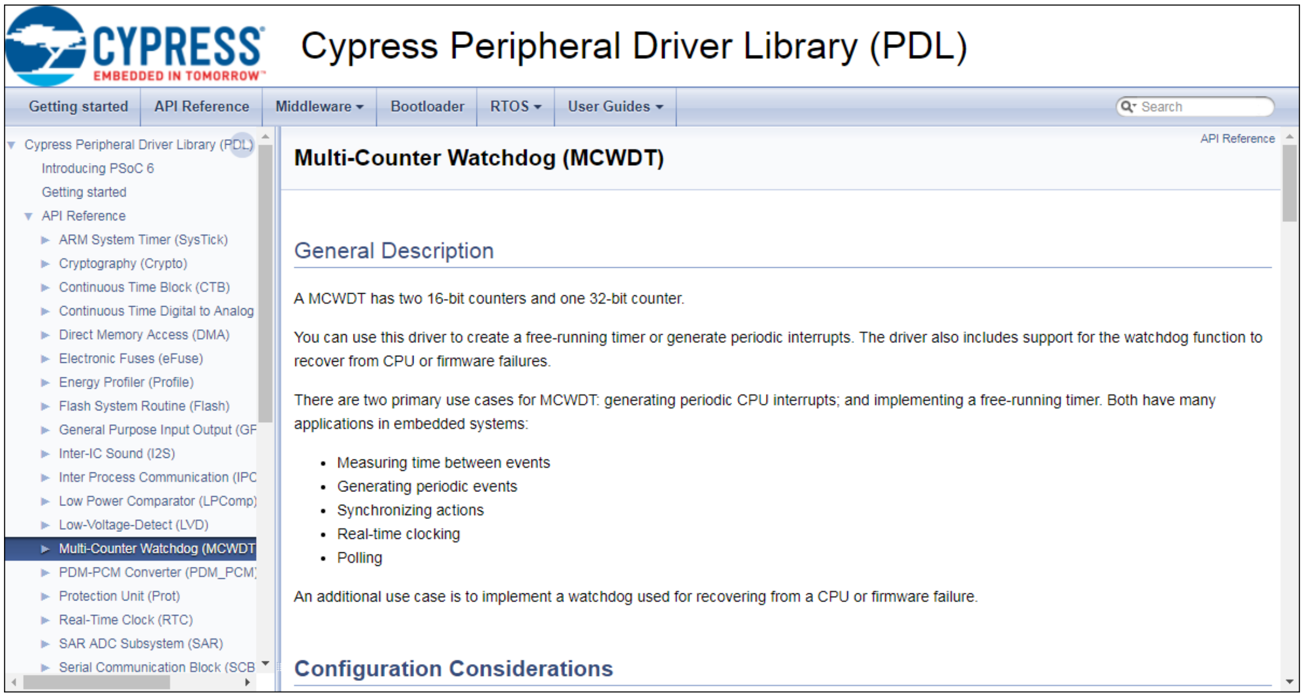
The *doc* folder also includes documentation for other software libraries included with the PDL, such as the BLE stack.

The *PDL API Reference* is HTML-based, and generated directly from source code.

In the API reference, use the navigation tree to find an overview of each peripheral, as well as technical details on macros, functions, and data structures. Figure 3 shows an example of the *PDL API Reference*. The actual documentation includes the installed PDL version number.

The PDL API documentation also has a search feature for defined terms (such as macros, constants, function names, and so forth). Enter a defined term in the search box, and the search function returns links to that term in the API documentation.

Figure 3. PDL Documentation


Cypress Peripheral Driver Library (PDL)
Getting started
API Reference
Middleware
Bootloader
RTOS
User Guides
Search
Cypress Peripheral Driver Library (PDL)
Introducing PSoC 6
Getting started
API Reference
ARM System Timer (SysTick)
Cryptography (Crypto)
Continuous Time Block (CTB)
Continuous Time Digital to Analog
Direct Memory Access (DMA)
Electronic Fuses (eFuse)
Energy Profiler (Profile)
Flash System Routine (Flash)
General Purpose Input Output (GF
Inter-IC Sound (I2S)
Inter Process Communication (IPC
Low Power Comparator (LPComp)
Low-Voltage-Detect (LVD)
Multi-Counter Watchdog (MCWDT
PDM-PCM Converter (PDM_PCM)
Protection Unit (Prot)
Real-Time Clock (RTC)
SAR ADC Subsystem (SAR)
Serial Communication Block (SCB
API Reference

Multi-Counter Watchdog (MCWDT)

General Description

A MCWDT has two 16-bit counters and one 32-bit counter.

You can use this driver to create a free-running timer or generate periodic interrupts. The driver also includes support for the watchdog function to recover from CPU or firmware failures.

There are two primary use cases for MCWDT: generating periodic CPU interrupts; and implementing a free-running timer. Both have many applications in embedded systems:

- Measuring time between events
- Generating periodic events
- Synchronizing actions
- Real-time clocking
- Polling

An additional use case is to implement a watchdog used for recovering from a CPU or firmware failure.

Configuration Considerations

3. Build and Run a PDL Project

In this section, you build and run an example PDL project using PSoC Creator. The goal is simply to ensure that the PDL is installed correctly, so you can compile, download, and run a code example.

Before You Begin

Download code example CE216795 - PSoC 6 MCU Dual-Core Basics. Unzip the package into a convenient location on your computer. The code example documentation lists the required tools:

- PSoC Creator v4.2
- PDL v3.0.x
- Hardware on which to run the code

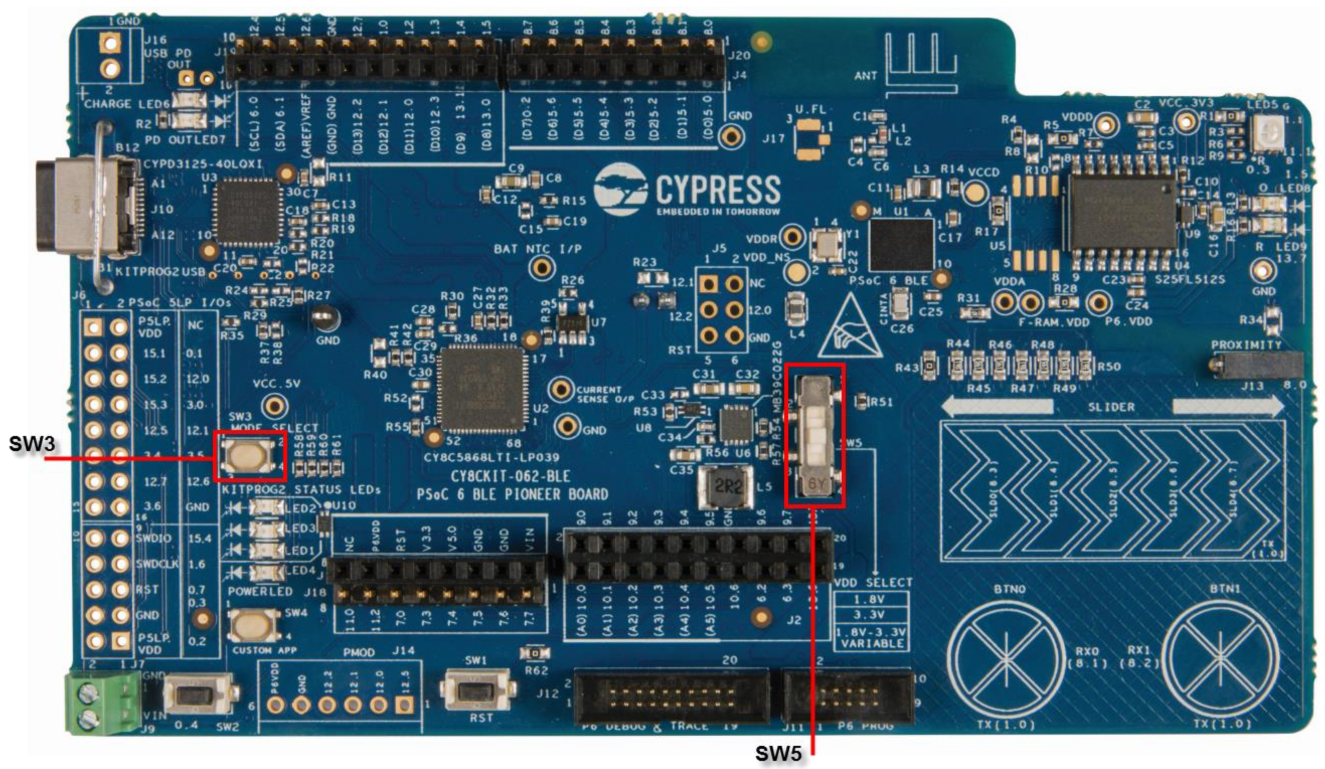
PSoC Creator 4.2 installs the PDL. A newer version of the PDL may be available at the PDL product page.

For hardware, this guide uses the PSoC 6 BLE Pioneer Kit, CY8CKIT-062-BLE. If you use other hardware, adjust the instructions accordingly.

This kit supports variable voltage. The code example requires that SW5 (VDD Select) is set to 3.3 V. This is the default configuration for the board.

Figure 4 shows the PSoC 6 BLE Pioneer Kit. If you run into problems, see the Troubleshooting section for some of the more likely causes and solutions.

Figure 4. PSoC 6 BLE Pioneer Kit



3.1 Building with PSoC Creator

These instructions assume that you are familiar with PSoC Creator. If you are not, there are several resources available to you. See the Related Resources section.

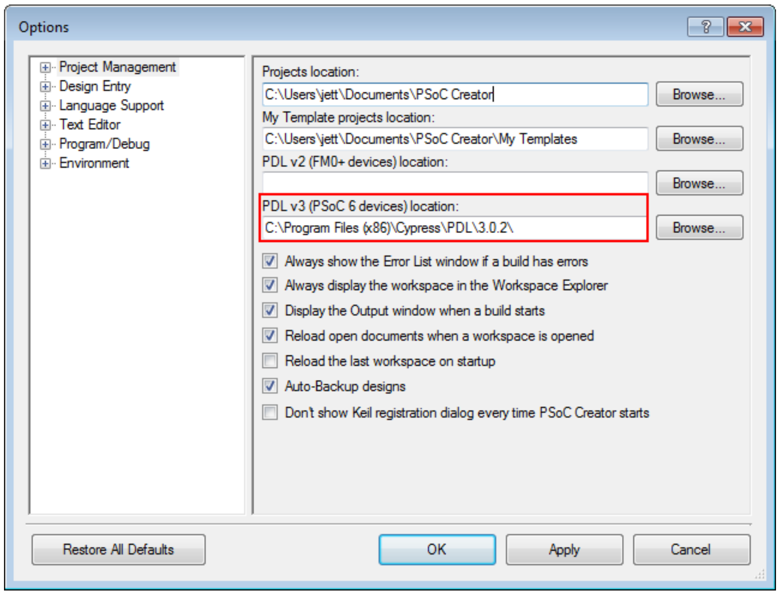
1. **Confirm that PSoC Creator can find the PDL code.**

   A. Select the **Tools** > **Options** command to open the **Project Management** panel.

   B. Confirm that the **PDL v3 (PSoC 6 devices) location** is correct.

   If the path listed is incorrect, browse to locate the PDL install directory. Figure 5 shows the default location.

Figure 5. Setting the PDL Location in PSoC Creator



Projects location: C:\Users\jett\Documents\PSoC Creator

My Template projects location: C:\Users\jett\Documents\PSoC Creator\My Templates

PDL v2 (FM0+ devices) location:

PDL v3 (PSoC 6 devices) location: C:\Program Files (x86)\Cypress\PDL\3.0.2\

- [✓] Always show the Error List window if a build has errors
- [✓] Always display the workspace in the Workspace Explorer
- [✓] Display the Output window when a build starts
- [✓] Reload open documents when a workspace is opened
- [ ] Reload the last workspace on startup
- [✓] Auto-Backup designs
- [ ] Don't show Keil registration dialog every time PSoC Creator starts

2. **Connect your board to your host computer.**

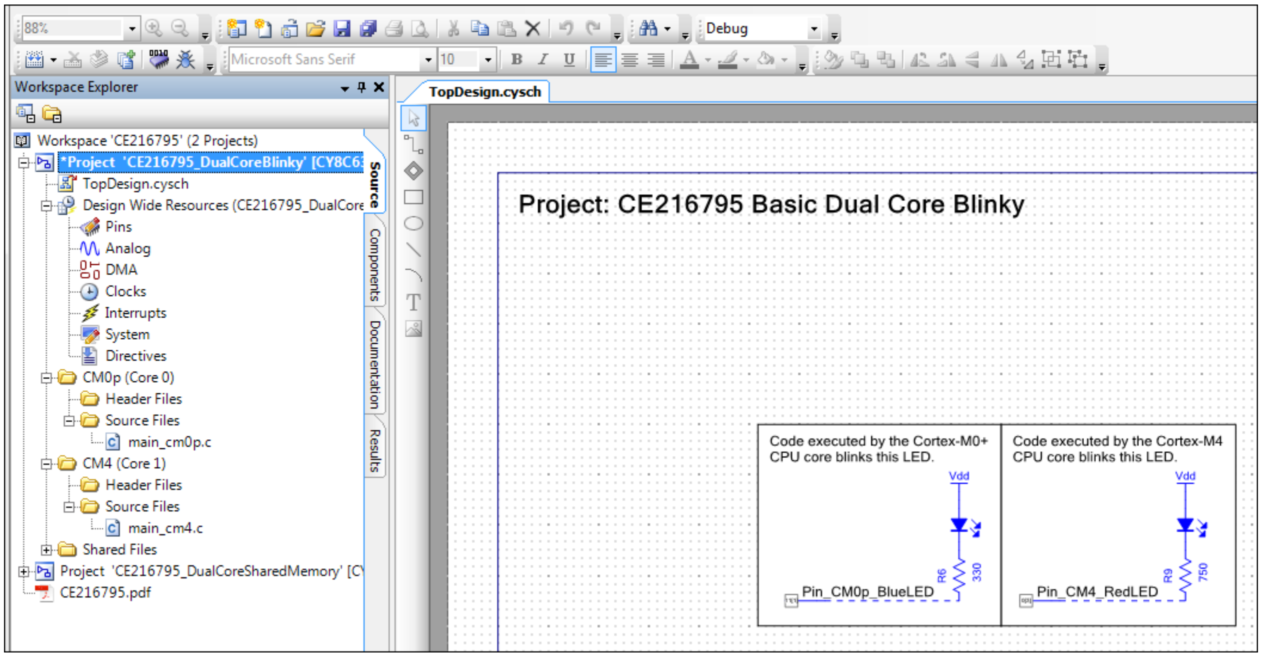
   Follow the instructions that came with your kit. The PSoC 6 BLE Pioneer kit includes the required USB Type-C cable.

3. **Open a code example.**

   The instructions use CE216795, and within that example, the 'DualCoreBlinky' project. If you have not already done so, download the code example from the web.

   Locate the *CE216795.cywrk* file and double-click it to open the workspace. The project appears in PSoC Creator.

Figure 6. The DualCoreBlinky Project


Workspace Explorer
Workspace 'CE216795' (2 Projects)
*Project 'CE216795_DualCoreBlinky' [CY8C6
TopDesign.cysch
Design Wide Resources (CE216795_DualCore
Pins
Analog
DMA
Clocks
Interrupts
System
Directives
CM0p (Core 0)
Header Files
Source Files
main_cm0p.c
CM4 (Core 1)
Header Files
Source Files
main_cm4.c
Shared Files
Project 'CE216795_DualCoreSharedMemory' [C
CE216795.pdf
Source
Components
Documentation
Results
TopDesign.cysch
Project: CE216795 Basic Dual Core Blinky
Code executed by the Cortex-M0+ CPU core blinks this LED.
Code executed by the Cortex-M4 CPU core blinks this LED.
Vdd
R6
330
Pin_CM0p_BlueLED
R9
750
Pin_CM4_RedLED

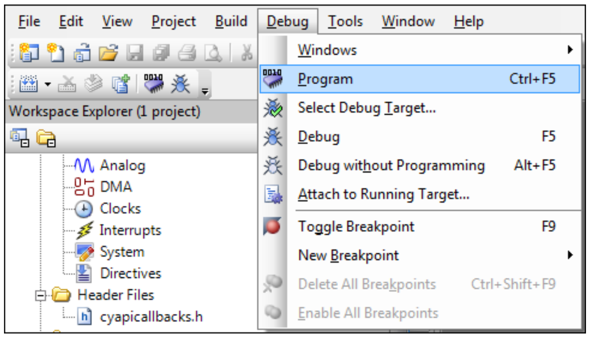
C. Read the code example documentation to understand what the example does, and how to observe results.

In this example, the CM0+ core drives the blue LED and the CM4 core drives the red LED. The firmware for each core uses a delay to control the flashing rate.

4. **Program the kit.**

A. Select **Debug** > **Program**. The **Select Debug Target** dialog appears.

Figure 7. Program the Kit

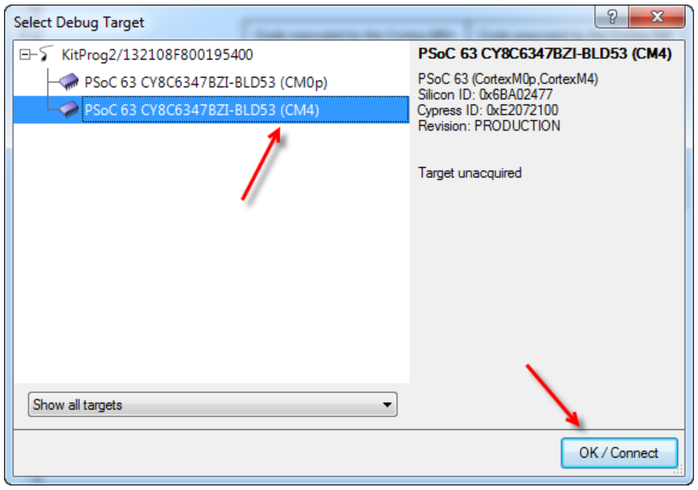


B. Select a target core.

Select the CM4 target, as shown in Figure 8, and click **OK/Connect**.

PSoC Creator debugs one core at a time, although in this simple exercise you do not run the debugger.

Figure 8. Select the Debug Target


Select Debug Target
KitProg2/132108F800195400
PSoC 63 CY8C6347BZI-BLD53 (CM0p)
PSoC 63 CY8C6347BZI-BLD53 (CM4)
Show all targets
OK / Connect

PSoC 63 CY8C6347BZI-BLD53 (CM4)
PSoC 63 (CortexM0p,CortexM4)
Silicon ID: 0x6BA02477
Cypress ID: 0xE2072100
Revision: PRODUCTION

Target unacquired

PSoC Creator generates the necessary source code for each core, compiles and links the code, downloads both the CM0+ and CM4 executables to the board, and runs the application.

5. **Observe the behavior described in the code example documentation.**

For the DualCoreBlinky project, the LED changes color among blue, red, purple, and off, based on the delay timing for each core.

Congratulations, you have successfully built a dual-core PDL-based application with PSoC Creator.

3.2 Troubleshooting

If you encounter an issue when you build and run a code example, you may find a solution here. See Figure 4 for the location of switches and jumpers referenced here.

Can't find a code example in the Find Code Example dialog.

- Ensure that you have selected the right device family.
- Use the **Filter by** text box to search for the code example.

Project doesn't open.

- Make sure that you are using PSoC Creator v4.2 or later for PSoC development.
- Launch PSoC Creator v4.2 first, and then try to open the project file or workspace.

Error on building code, "Unable to find required PDSC file".

- The path to the PDL is incorrect. Select the **Tools** > **Options** command and browse to the correct location.

Board isn't getting power.

- Ensure that the USB cable is attached to both the board and the host computer.
- Try a different USB cable.

LEDs are dim or not functioning as expected.

- Ensure that SW5 (VDD Select) is set to 3.3 V.

Cannot acquire the board using a CMSIS-DAP connection.

- PSoC Creator uses the KitProg2 connection. Press SW3 on the PSoC 6 BLE Pioneer Kit to toggle the debug connection between KitProg2 and CMSIS-DAP. If LED2 is on, the board is in KitProg2 mode.

Cannot connect to the board (COM port not working).

- Make sure that no other IDE instance is connected to the board (from another code example perhaps).
- Make sure any terminal program is closed. (Sometimes they take ownership of the COM port.)
- Unplug the USB cable from the PC and plug it back in.
- Plug the cable into a different USB connector on your PC.
- Use Device Manager to ensure that there is a COM port for the board.
- If all else fails, reboot to clear any COM port conflicts that might occur in Windows.

Execution does not stop at the first line of main().

- This is expected behavior. The PSoC Creator **Debug** > **Program** command does not launch the debugger.
- If you want to debug, choose **Debug** > **Debug**. Execution stops at the first line of `main()` on the targeted core.

4. Developing Code Using the PDL

This section introduces PDL fundamentals such as working with registers, configuring a peripheral, managing pins, and other topics related to developing code using the PDL.

As in any embedded development project, to develop software for a PSoC 6 MCU device you are responsible for the design and implementation of the firmware, including:

- Configuring and initializing clocks
- Identifying the correct clock source for each peripheral
- Pin selection – which pins are used by which peripherals
- Pin configuration
- Trigger routing – which pins act as inputs to which peripherals
- Peripheral selection – when multiple instances of a peripheral are available, which is required
- Peripheral configuration – declaring and populating all required configuration structures
- Handling power mode transitions

The PDL provides APIs or source files to assist with all of these tasks, as detailed in this section of the User Guide.

In addition to the usual tasks for embedded firmware development, the dual-core architecture of the PSoC 6 MCU device introduces additional tasks. See Developing Dual-Core Applications. For example, firmware on a dual-core architecture may need to handle inter-processor communication. See Managing Inter-Processor Communication.

You may prefer to begin development with a pre-configured template project provided by the PDL. See Debugging a Dual-Core Application.

You can use the PDL effectively with any suitable embedded development environment. This section of the User Guide discusses PDL development without regard to any particular IDE.

However, PSoC Creator makes some of these tasks completely transparent, and makes others easier. See Using the PDL with PSoC Creator.

4.1 Design and Common Coding Constructs

A PSoC 6 MCU device has a set of hardware blocks (peripherals). The set of available peripherals varies per device.

Each peripheral has a set of registers that control that peripheral's behavior. There may be multiple instances of a peripheral; for example, the Serial Communication Block (SCB). If there are multiple instances, each instance has its own register set. Each register set has a base hardware address that points to all the registers for that instance of that peripheral.

Each peripheral and instance is configurable. You modify values in a configuration structure to change behavior to fit your requirements. The PDL manages register access using the base hardware address.

Finally, a single peripheral instance (however configured) may work on multiple instances of data. For example, an SCB configured as a UART operates on data buffers, and maintains status information about those buffers.

To enable multiple instances of configurable peripherals operating on (potentially) multiple instances of data, the PDL implements some coding constructs to standardize what you need to do to use the PDL.

There are three common parameters that appear in many PDL API function calls, as appropriate. For example, this is the prototype for the function that initializes the SCB configured as a UART (ignoring the return value).

```
Cy_SCB_UART_Init (CySCB_Type *                    base,
                  cy_stc_scb_uart_config_t const *  config,
                  cy_stc_scb_uart_context_t *       context)
```

Table 3 describes each of these common parameters, with links to a more in-depth discussion for each.

Table 3. Common PDL Parameters

Parameter	Data Type	Purpose	See
`base`	`Cy<peripheral>_Type*`	Base hardware address for an instance of a peripheral	Finding the Base Hardware Address
`config`	`cy_stc_<peripheral>_config_t*`	The address of a structure with configuration information for an instance of a peripheral	Configuring a Peripheral
`context`	`cy_stc_<peripheral>_context_t*`	The address of memory allocated for the driver's use	Providing Context for a Peripheral

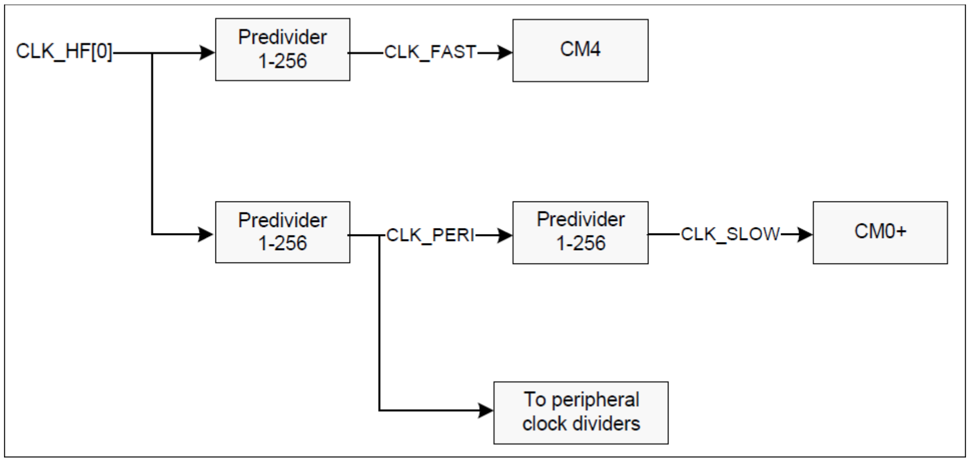
Any given PDL API function call may use one or more of these parameters, as well as other parameters required for the function. These three parameters are named consistently throughout the API.

4.2 Configuring Clocks

A PSoC 6 MCU device has internal and external clock sources, and generated clocks such as a frequency-locked loop (FLL), or phase-locked loop (PLL). When creating an application for PSoC 6 MCU, you need to enable and configure the source clocks and clock dividers for your design. The clocks and clock tree are fully documented in the *Clocking System* chapter of the *Technical Reference Manual* (TRM).

A peripheral can only be clocked by a peripheral clock divider. CLK_PERI is the source clock for all programmable peripheral clock dividers, as shown in Figure 9. The output of any divider can be routed to any peripheral.

Figure 9. The Peripheral Clock Tree



The PDL provides the System Clock (SysClk) driver that you can use to configure the clocking system for your design. For example, there is a function call to set the divider for the peripheral clock tree, and others to set, get, enable, or disable individual clock dividers.

4.3 Configuring Interrupts

Interrupts are set up at the system level for any peripheral that handles an interrupt.

A PSoC 6 MCU device supports many system and peripheral interrupts. These are processed by the nested vector interrupt controller (NVIC) of the individual core. On the CM4 core, system interrupt source 'n' is directly connected to the corresponding IRQn. The CM0+ core has only 32 IRQs. Any of the interrupt sources can be connected to any IRQn on the CM0+ core. See the *Interrupts* chapter of the Technical Reference Manual.

The PDL provides the System Interrupt (SysInt) driver to assist in configuring interrupts. To connect an ISR to an interrupt signal, you initialize the system interrupt with a configuration structure (containing the interrupt number and priority), and provide the address of the ISR. For the CM0+ core, you provide the interrupt source as well as the IRQn (because of interrupt muxing). Some CM0+ IRQs are reserved for the system. See the SysInt driver section of the PDL API Reference. Various drivers have support in their individual APIs for setting, clearing, and masking interrupts, or to specify a callback routine.

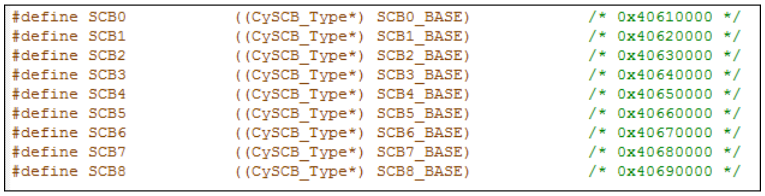
4.4 Finding the Base Hardware Address

Many PDL API function calls require the base hardware address for the peripheral instance.

A device-specific header file defines the symbols for the base hardware address of each instance of each peripheral. For example, the PSoC 6 BLE Pioneer kit uses the cy8c637bzi_bld74 device. The PDL includes the device-specific header file in the *devices* folder: *<PDL directory>\devices\psoc6\psoc63\include\cy8c637bzi_bld74.h*.

Each such file defines a symbol that represents a pointer to the base hardware address for each instance of each peripheral on the device. Use this symbol in your code. Figure 10 shows the symbols for instances of the SCB peripheral on this device. This device supports nine SCB instances. The header file contains similar definitions for each peripheral.

Figure 10. Symbols for SCB Base Hardware Addresses



```
#define SCB0          ((CySCB_Type*) SCB0_BASE)          /* 0x40610000 */
#define SCB1          ((CySCB_Type*) SCB1_BASE)          /* 0x40620000 */
#define SCB2          ((CySCB_Type*) SCB2_BASE)          /* 0x40630000 */
#define SCB3          ((CySCB_Type*) SCB3_BASE)          /* 0x40640000 */
#define SCB4          ((CySCB_Type*) SCB4_BASE)          /* 0x40650000 */
#define SCB5          ((CySCB_Type*) SCB5_BASE)          /* 0x40660000 */
#define SCB6          ((CySCB_Type*) SCB6_BASE)          /* 0x40670000 */
#define SCB7          ((CySCB_Type*) SCB7_BASE)          /* 0x40680000 */
#define SCB8          ((CySCB_Type*) SCB8_BASE)          /* 0x40690000 */
```

For example, this is the prototype for the function to enable the 'Clear to Send' input for the UART.

```
void Cy_SCB_UART_EnableCts (CySCB_Type * base)
```

Assuming that your design uses the SCB instance three configured as a UART, your code might look like this:

```
Cy_SCB_UART_EnableCts (SCB3);  // Enable clear to send
```

The PDL uses the base hardware address to access the necessary registers, so you do not need to know about or reference specific registers. You can access a bit or register directly from your code. See Accessing Registers Directly.

4.5 Configuring a Peripheral

The PDL defines all necessary configuration structures. Specific structures and fields vary for each peripheral. For more complex peripherals, some fields within the configuration structure may be the address of other configuration structures. When you initialize an instance of a peripheral, you must provide the required configuration structure with all fields filled in to specify your desired behavior. (Some peripherals do not require a configuration structure.)

Configuration structures are defined in a peripheral-specific header file, found in the PDL *drivers* folder. For example, the structures for the SCB configured as a UART are defined in this file: *<PDL directory>\drivers\peripheral\ scb\cy_scb_uart.h*.

Use the *PDL API Reference* for details on structures, field names, data types, and the range of valid values. Figure 11 shows a portion of the documentation for the UART configuration structure.

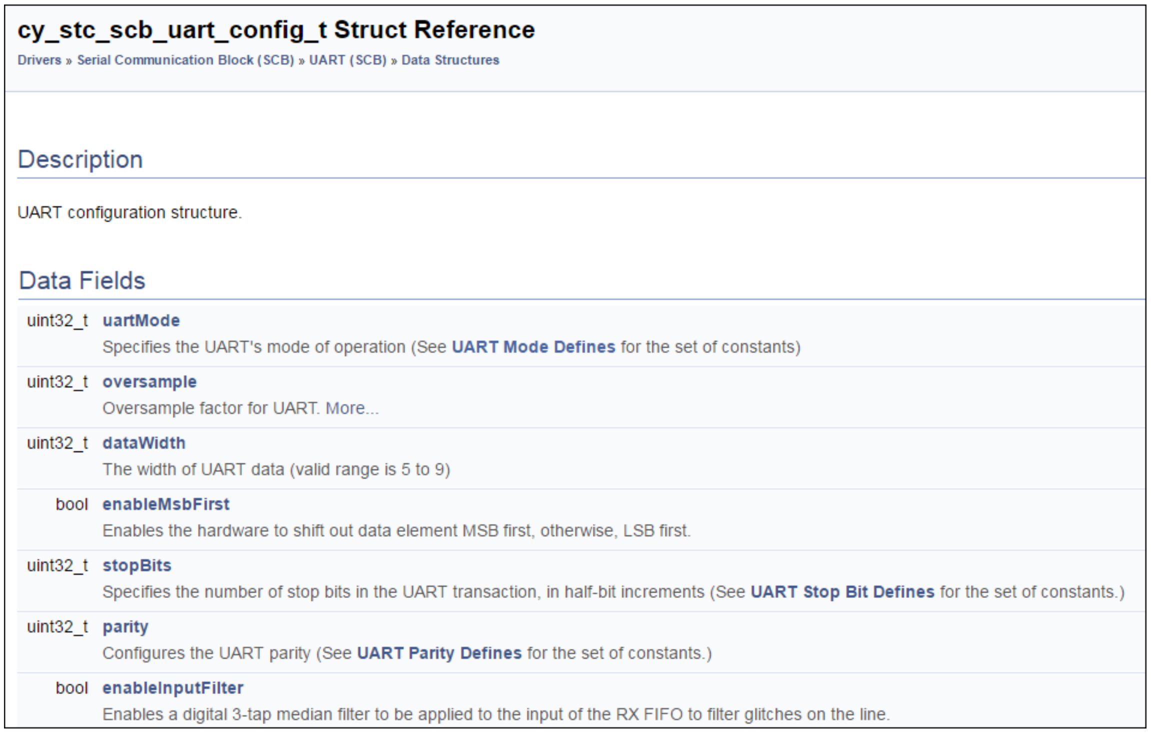
To configure a peripheral, instantiate the necessary configuration structure, and fill in the required values to define the behavior of the peripheral. For example, for the SCB peripheral configured as a UART, you might have code that declares a configuration structure named `myUART_config`.

```
cy_stc_scb_uart_config_t myUART_config;  // UART configuration structure
```

After filling the fields, use the address of this structure in the call to initialize the peripheral.

```
result = Cy_SCB_UART_Init (SCB3, &myUART_config, &myUART_context);
```

Figure 11. Documentation for the UART Configuration Structure



cy_stc_scb_uart_config_t Struct Reference

Drivers » Serial Communication Block (SCB) » UART (SCB) » Data Structures

Description

UART configuration structure.

Data Fields

uint32_t	**uartMode** Specifies the UART's mode of operation (See **UART Mode Defines** for the set of constants)
uint32_t	**oversample** Oversample factor for UART. More...
uint32_t	**dataWidth** The width of UART data (valid range is 5 to 9)
bool	**enableMsbFirst** Enables the hardware to shift out data element MSB first, otherwise, LSB first.
uint32_t	**stopBits** Specifies the number of stop bits in the UART transaction, in half-bit increments (See **UART Stop Bit Defines** for the set of constants.)
uint32_t	**parity** Configures the UART parity (See **UART Parity Defines** for the set of constants.)
bool	**enableInputFilter** Enables a digital 3-tap median filter to be applied to the input of the RX FIFO to filter glitches on the line.

4.6 Providing Context for a Peripheral

A PDL driver may require memory to maintain status information, perform calculations, or generate results. Such drivers do not allocate memory for the data on which they operate. Firmware allocates the required memory.

Firmware allocates memory by declaring a `context` structure, and passes the address of that structure in function calls. That is the only way in which firmware uses a context. The PDL manages the contents of the context structure. Firmware does not write or read the values in the context structure. In effect, the context is a scratch pad in memory for the PDL driver to do its work.

Firmware must ensure that a context variable remains in scope when in use. Typically a context variable is declared as `static`, or as a global variable.

The PDL defines all necessary context structures in a peripheral-specific header file, found in the PDL *drivers* folder. This is the same file that defines configuration structures. For example, the structures for the SCB configured as a UART are defined in this file: *<PDL directory>\drivers\peripheral\scb\cy_scb_uart.h*.

When working with a UART, firmware declares a context structure, perhaps named `myUART_context`.

```
static cy_stc_scb_uart_context_t   myUART_context;  // UART context structure
```

Firmware simply passes the address of this structure in function calls. For example, to terminate a UART receive operation, your code might look like this:

```
Cy_SCB_UART_AbortReceive (SCB3, &myUART_context);  // For the SCB3 UART
```

4.7 Initializing a Peripheral

Use the `Init()` function in the form `Cy_<peripheral>_Init()`.

The parameters required for the `Init()` function will vary per peripheral, but may include one or more of: the base hardware address, the address of a configuration structure, and the address of a context structure.

For example, to initialize the Watchdog Timer (WDT) peripheral, no parameters are required. Your code might look like this:

```
Cy_WDT_Init ();
```

By contrast, to initialize an SCB configured as a UART, you provide all three parameters.

```
result = Cy_SCB_UART_Init (SCB3, &myUART_config, &myUART_context);
```

Initializing a peripheral sets various registers to the values in the configuration structure. If no configuration structure is required, it sets the registers to their defined initial state.

4.8 Enabling a Peripheral

After initializing, you typically must enable the peripheral. Most peripherals have an `Enable()` function in the form `Cy_<peripheral>_Enable()`.

The parameters required for the `Enable()` function will vary per peripheral. If there are multiple instances of the peripheral you typically provide the base hardware address. Any particular peripheral may require other parameters. See the *PDL API Reference* for guidance.

For example, to enable the WDT, your code would look like this:

```
Cy_WDT_Enable ();   // Enable the watchdog
```

To enable a UART, you provide the base hardware address. Your code might look like this:

```
Cy_SCB_UART_Enable (SCB3); // Enable the SCB3 block
```

4.9 Using a Peripheral

After you have initialized and enabled a peripheral, you can use it.

To use the peripheral, make PDL API function calls. For example, you may need to start a timer or clear an interrupt. See the *PDL API Reference* for guidance on function calls available for any peripheral, and what parameters are required.

For example, to start a UART receive operation using the SCB3 instance, your code might look like this:

```
result = Cy_SCB_UART_Receive (SCB3, &myBuffer, size, &myUART_context);
```

4.10 Managing Inter-Processor Communication

In a dual-core system, care must be taken to ensure that the two cores don't access the same resource simultaneously. You can implement a semaphore and mutex design in your firmware as required. The IPC hardware block provides the functionality for multiple processors to communicate and synchronize their activities. See the 'Inter-Processor Communication' chapter of the TRM.

The PDL provides the Inter-Processor Communication (IPC) driver to enable this. The PDL IPC driver implements the concepts of a semaphore, and a pipe. To use the IPC, you take five steps. The PDL API provides functions for each step in the process.

- The source processor acquires the pipe, which locks the pipe
- The source processor puts data in the pipe
- The source processor notifies the destination processor
- The destination processor retrieves and processes the data
- The destination processor releases the pipe when done (which notifies the source processor)

The data in the pipe is a 32-bit void pointer, which can point to data of arbitrary complexity.

4.11 Managing Power Mode Transitions

The System Power Management driver (SysPm) implements a callback mechanism to handle power mode transitions. This mechanism is fully documented in the *PDL API Reference*.

This mechanism allows any driver to perform any operation required to prepare for the specific power mode, or to abort the transition. The power mode transition callbacks are maintained as a linked list. The driver calls them in order, either first to last, or last to first, as appropriate.

Every driver concerned with entering or returning from a low-power state *must* register callback functions with SysPm. If there is any order dependence among the drivers in your firmware, register the callbacks in the order you require. For example, callbacks that configure global resources, such as clock frequencies, should be registered last.

If your firmware does not register any SysPm callbacks, the device just executes the power mode transition.

There are four power mode events that the callback must handle as appropriate.

Table 4. Handling Power Mode Transitions

Event	Meaning	Call Order	Action
CHECK_READY	Is the driver ready?	First to last	Prepare for transition to the specific power mode, or return a fail message to abort the transition.
CHECK_FAIL	A driver has aborted the transition.	Last to first*	Undo any preparation performed in response to the ready message.
BEFORE_TRANSITION	All drivers are prepared for transition.	First to last	When all callbacks have executed, the device enters the specific power mode.
AFTER_TRANSITION	The device has returned from a low-power mode	Last to first	Perform any task required upon wakeup.

* Starting with the callback just prior to the failure

There are some exceptions to this process based on particular transitions. For example, the device is rebooted on wakeup from the Hibernate power mode, so the `AFTER_TRANSITION` event does not occur. See the SysPm documentation in the *PDL API Reference* for details.

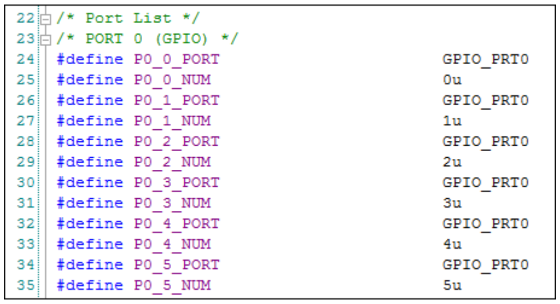
4.12 Managing Pins

Which pin to use for which input or output for a peripheral is based on the hardware design for the target system. Your software may need to manage the state of individual pins, based on the system's pin assignments.

The PDL provides the General Purpose Input Output (GPIO) driver to assist. The GPIO driver provides an API so that software can configure and access pins based on port and pin number. You can initialize, read, write, set, clear, or invert the state of a pin, get or clear the interrupt state on the pin, and more. See the *PDL API Reference* for details on the GPIO driver.

A typical GPIO function call requires both the port hardware address and the specific pin. The PDL provides a package-specific GPIO header file to define symbols for each port and pin. For example, the file *gpio_psoc63_116_bga_ble.h* defines these symbols for the device on the PSoC 6 BLE Pioneer Kit. Figure 12 shows some of the symbols defined in that file. The symbol name takes the form Px_y where x is the port, and y is the pin.

Figure 12. Symbols for Port and Pin



```
/* Port List */
/* PORT 0 (GPIO) */
#define P0_0_PORT                         GPIO_PRT0
#define P0_0_NUM                          0u
#define P0_1_PORT                         GPIO_PRT0
#define P0_1_NUM                          1u
#define P0_2_PORT                         GPIO_PRT0
#define P0_2_NUM                          2u
#define P0_3_PORT                         GPIO_PRT0
#define P0_3_NUM                          3u
#define P0_4_PORT                         GPIO_PRT0
#define P0_4_NUM                          4u
#define P0_5_PORT                         GPIO_PRT0
#define P0_5_NUM                          5u
```

For example, the prototype for the call to invert the output signal on a pin looks like this:

```
void Cy_GPIO_Inv  (GPIO_PRT_Type * base, uint32_t pinNum )
```

Your code might look like this:

```
Cy_GPIO_Inv (P0_3_PORT, P0_3_NUM); // Invert output signal on port 0 pin 3
```

If you need to determine a port address at runtime, use the `Cy_GPIO_PortToAddr()` function.

4.13 Trigger Muxing

Your design may require that the output from one peripheral triggers behavior in one or more other peripherals. For example, a SPI (SCB) peripheral block receives data. After the data is received, a signal from the SCB triggers a DMA channel to transfer the received data to a memory buffer.

The PSoC 6 MCU trigger multiplexer block implements a collection of multiplexers that can connect any trigger signal from any peripheral to any other peripheral. To support this, the hardware implements an input layer and an output layer. The input layer has reduction multiplexer groups, which have as input the trigger signals coming from different peripheral blocks. The reduction multiplexer routes these signals to an intermediate trigger signal. The output side uses that intermediate trigger as its input, and uses a distribution multiplexer to send that signal to one or more peripherals. The design also supports a software trigger for any signal. See the *Trigger Multiplexer Block* chapter of the TRM.

The PDL provides the Trigger Multiplexer (TrigMux) driver to enable this. To route a trigger signal from one peripheral in the PSoC to another, you configure both a reduction multiplexer and a distribution multiplexer. In effect, you define where the signal comes from, and where it goes to. The software trigger will trigger all peripherals connected to that output signal.

4.14 Accessing Registers Directly

The PDL simplifies the embedded programming process by providing a high-level API. You may wish to work at the register level directly. Working at that level requires knowledge of registers and bits that control the features you want to use.

Studying the PDL source code is a useful way to approach the detailed knowledge required to program a microcontroller at the register level. Identify the API function that provides the behavior you want to implement. Then examine the source code to identify the registers and bit fields that the PDL manipulates. This approach gives you a place to start learning. Here's a simple example.

Suppose you want to modify compare value zero of a PWM to adjust its duty cycle, by accessing the register directly.

1. **Identify the API function call that does what you want.**

   In this case, the PDL call is `Cy_TCPWM_PWM_SetCompare0().`

2. **Locate the source code for that function.**

   The PDL naming conventions tell you the source code is in a file named *cy_tcpwm_pwm.c*. In many IDEs, you can right-click the function name and jump to the function definition.

3. **Examine the code.**

   Here is the implementation for this function from the PDL source code.

```
void Cy_TCPWM_PWM_SetCompare0(TCPWM_Type *base, uint32_t cntNum, uint32_t compare0)
{
    base->CNT[cntNum].CC = compare0;
}
```

The input parameters specify the peripheral base address, the counter to modify, and the new compare value for that counter. It then sets the value in the correct register.

Table 5. Parsing PDL Source Code for Low-level Programming

Base Address	Which Set of Registers	Register	Value
base	CNT[cntNum]	CC	The compare value

You can use this information to locate details in the Technical Reference Manual.

In your own code, you may want to emulate how the PDL works. However, the library's approach may not be optimal for your circumstances. You can use it as an example of one approach to getting a task done.

4.15 Developing Dual-Core Applications

The PSoC 6 MCU architecture includes dual-core devices with ARM Cortex-M4 and Cortex-M0+ microcontrollers. Firmware is written for and runs on only one core. However, a single downloadable application image can include firmware for each core, hence a dual-core application. The firmware on each core can communicate with the firmware on the other core. PSoC 6 MCU devices have a single memory map, so the cores can share system resources. Most PDL peripherals can be used by either core. See the *PDL API Reference* for any limitations.

How you structure your development projects to create a dual-core application will depend upon your IDE. You may need a separate project for each core. Some IDEs support managing multiple projects in a single workspace. However, each project is typically fully independent, and must contain all files required to build the executable for that core.

Because the cores share the same memory map, each executable should reside at a separate address in memory space. The PDL provides template projects for supported IDEs. These projects include linker files to accomplish this task. You can find the linker files for supported IDEs in the *devices* folder for each series, on this path:

<PDL install directory>\devices\psoc6\psoc63\common\

There is a folder for each supported IDE.

In the template project for each supported IDE, post-build commands invoke the CyMCUElfTool. This tool processes the linked ELF images to ensure that memory regions either do not overlap, or contain identical bytes. It adds data necessary for the boot process, performs security checking, and merges images for multiple cores into a complete image for an entire application. This happens after the CM4 project is built, so you must build the CM0+ application first, and then build the CM4 application.

The PSoC 6 MCU architecture includes some devices where the CM0+ core is not accessible to users. In this case the Flash Boot handles setup of the CM0+ core and starts the Cortex-M4 core. See the *PDL API Reference* topic on *System Configuration Files* for information on device initialization, memory definition, and heap configuration.

4.15.1 Debugging a Dual-Core Application

Some tools support dual-core debugging, meaning the debugger can connect to the application on each core simultaneously. For those tools, the default template projects are configured for dual-core debugging. As noted, the post build commands run after the CM4 project is built. The following configuration options are set in the default projects. The precise settings will vary per IDE.

- Dual-core debugging is enabled
- The CM0+ project does *not* download the executable to the board
- The CM4 project builds and downloads the combined executable image to the board

Note: You must ensure that the CM4 project actually builds! Some tools do not establish a dependency between the CM4 and CM0+ projects. If you modify only the CM0+ project, the CM4 project is not rebuilt, so does not merge the modified CM0+ executable into the final image, and you end up with incorrect CM0+ code running. It is always best to force a clean and rebuild of both the CM0+ and the CM4 projects, to ensure an accurate final image is downloaded to the board.

If you wish to do single-core debugging, then modify the default settings in the template project for your IDE. Again, precisely what settings you modify will vary per IDE. However, in general you should:

- Disable dual core debugging support.
- Ensure that the CM0+ project downloads its own executable.

4.16 Building a Secure System

The definition of a secure system can vary depending on the application. It can range from not allowing firmware to be altered to not allowing anyone to examine any internal firmware or communicate with the device without possessing the correct security key. With the PSoC 6 MCU, you can create as secure a system as required.

A secure system requires a chain of trust. It starts with initial boot code stored in ROM that cannot be changed. Each step of the boot process builds upon that: confirming that the previous step is valid and trustworthy using digital signatures, and public and private cryptographic keys. If a change is detected, the boot process can be aborted.

The PDL provides an example project that builds a secure system. The example is in the *<PDL install directory>\security* folder. A PSoC Creator project is provided, along with all source code.

4.17 Using PDL Template Projects

Template projects are provided for each device series for each supported IDE. There is a project file for each core (CM0+ and CM4). For example, to target a PSoC 63 device, you find the template project in *<PDL install directory>**devices**psoc6**psoc63**projects*\.

Each project file has targets for each device in the series. Each target in each project uses the correct startup and configuration files for the targeted device.

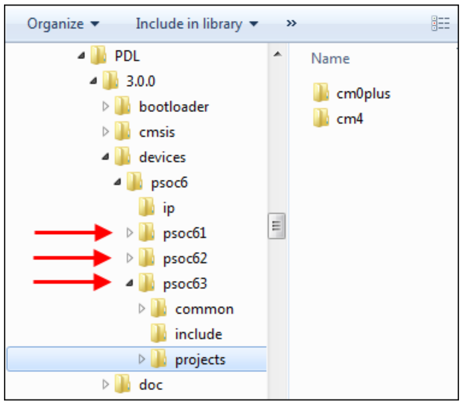
You can use a template project as a starting point for your development. A template project is a generic project that you customize for your development. For example, the template project:

- has multiple build targets (a build and release version for each specific device in a series)
- does not include PDL source files, but has include paths to the commonly-used PDL header files
- specifies the generic CM0+ or CM4 core, not a specific device
- uses device-specific linker command files and compiler options in each build target
- may not use your preferred debug connection
- may use a device-specific flash configuration file

You will need to modify particular settings, options, and files for your project. Because a template project has device-specific builds, many default settings in each build are likely to be acceptable. For example, the preprocessor definition for the specific part number is set by default in each build target.

Device-specific and IDE-specific files are in the *<PDL directory>**devices**psoc6*\ folder.

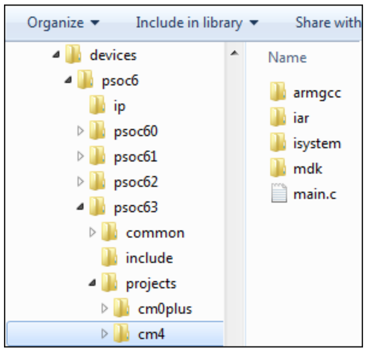
Figure 13. Locating Device-Specific Files



Within the directory for a specific series*,* the *common* folder has system initialization files. It also contains IDE-specific startup and configuration files. The *include* folder contains the device-specific header files.

The *projects* folder has subfolders for each PSoC 6 core (Cortex M0+ and M4). For each core, there is an empty *main.c* file. Each IDE folder has a project file. Each project file has build targets for each specific device in the series. Each target is configured to use the correct startup and configuration files, and the empty *main.c* file.

Figure 14. The *projects* Folder Structure



If you debug your template project with the µVision IDE, you may need to install special debug support on some platforms (such as Windows 10). You add support for these probes by installing the *cypress.uvision_support.1.0.0.pack*. The pack is located here:

<PDL 3.x install folder>\tools\probes\cypress.uvision_support.1.0.0.pack

To install the pack, locate the file and double-click. The Pack Installer installs the necessary files in the µVision IDE. See Enabling Debug Probes in µVision IDE for details.

The template project does not include PDL source files, because you will use a unique set of files in your project.

What you do to create a custom project depends upon your particular work flow. One reasonable approach is.

1. **Make a copy of the particular device series folder.**

   For example, make a copy of the *psoc63* folder. This contains everything you need to get started. In these steps, we will refer to this copy as “your project folder”. It contains your project file, already set up for your device. Keep your project folder in the *psoc6* folder, beside the original so that paths to header and source files continue to work.

   You can relocate your project folder if you wish, but if you do, you must update include paths. You may also need to remove and add the source files already in the project file. An IDE may store a source file location as an absolute rather than a project-relative path. If you move the folder, absolute paths will break.

   You may rename your project folder if you wish.

   You may delete any IDE-specific folders you don’t use. You may also remove build targets from the project file for any device that you aren’t using.

2. **Add PDL source files to the project as required for your firmware.**

   From your project folder, use either or both of the CM0+ or CM4 projects for your preferred IDE. Add the files you need from the PDL *drivers, middleware*, *rtos,* or *utilities* folders. Which files you require depends upon your hardware design and which features of the PDL you use.

   If you want to relocate your project folder, do that before adding PDL source files.

3. **Add or update include paths to required header files.**

   Precisely how to do this will vary based on your circumstances. We assume that you know how to add or edit paths in your preferred IDE.

   The paths you must add or update depend upon whether you relocated your project folder, and the header files that you use. Typically, you may need to add or update these paths:

   - Update existing paths if you relocated your project folder (e.g., *cmsis\include* and *drivers\peripheral*)
   - Add paths to the PDL *utilities*, *middleware*, or *rtos* header files if you use them

If path-related errors continue when you build your code, add or modify the paths until your IDE can locate all the header files.

4. **Include required header files.**

   Write code to include required header files. A good approach is to create a master header file that includes all required headers. In *main.c*, you can include the master header file. For example, PSoC Creator generates a file named *project.h* for this purpose.

5. **Write your firmware.**

   Write code to configure, initialize, enable, and use each peripheral. This process is the same in any development environment. The PDL defines a function API for each peripheral. Use the *PDL API Reference* to learn about each peripheral's API. See Using the PDL with PSoC Creator to learn how it simplifies this process.

4.18 PDL Naming Conventions

The PDL code uses certain naming conventions that help you more easily identify what a variable is, or how to use it effectively. All PDL symbols begin with the prefix `cy_`, capitalized as appropriate for the named item. Table 6 explains the common naming conventions.

Table 6. PDL Naming Conventions

Naming Convention	Example	Description
Prefix `cy_stc_`	`cy_stc_rtc_config`	This is a data structure.
Postfix `_t`	`cy_stc_rtc_config_t`	This is a data type, typically associated with a structure or enumeration.
Prefix `cy_en_`	`cy_en_rtc_alarm`	This is an enumeration. A parameter in a function call may require an enumerated type. This tells you that there are enumerations you should use to set the value for that field.
`Cy_<Peripheral>_<Function Name>()`	`Cy_RTC_GetDateAndTime()`	This is a function call. It includes the acronym for the peripheral, followed by the actual function name.
`<PERIPHERAL>_Type`	`TCPWM_Type`	The struct type definition for the hardware register set contained in the block. See Finding the Base Hardware Address.
`Parameter name base`	`Cy_MCWDT_Init (base, config)`	The base hardware address. See Finding the Base Hardware Address.
Parameter or term containing “config”	`cy_stc_rtc_config_t`	This item relates to a configuration structure to specify peripheral behavior and features. See Configuring a Peripheral.
Parameter or term containing “context”	`cy_stc_scb_uart_context_t`	This item relates to a context structure, which is user-allocated memory for the peripheral to use. See Providing Context for a Peripheral.

5. Using the PDL with PSoC Creator

The PDL is a stand-alone software development kit for PSoC 6 MCU devices. It is also fully integrated with the PSoC Creator Integrated Design Environment. This document assumes that you are familiar with PSoC Creator. If you are not, there are several resources available to you. See the Related Resources section.

PSoC Creator simplifies PDL development by generating symbols, configuration structures, and a simplified function API based on the Components in your design. You can use these resources in your code. The symbols, structures, and API generated by PSoC Creator are fully compatible with the PDL.

PSoC Creator also makes some system design tasks, such as setting up clocks or assigning pins, significantly easier by providing an intuitive UI.

In addition, PSoC Creator enables access to programming the Universal Digital Blocks (UDB) on the PSoC 6 MCU device. A UDB contains programmable logic devices that enable you to build logic at the hardware level that links other hardware in your design, or to create a stand-alone block that performs a new function. See the PSoC Creator Component Author Guide to learn more about UDB-based Components.

PSoC Creator implements Components for most PDL peripherals. In PSoC Creator, you configure Components and generate code, rather than writing configuration code yourself.

5.1 Configuring Clocks with PSoC Creator

PSoC Creator provides a user interface for managing clocks. This allows you to see each clock in the tree, and configure it. In addition, when you drag a clock component onto the design schematic and connect it to a peripheral's clock terminal, you are configuring the peripheral clock divider for the corresponding peripheral. PSoC Creator generates the required calls to the PDL System Clock API based upon your design. See Configuring Clocks.

5.2 Configuring Interrupts with PSoC Creator

PSoC Creator has a user interface for configuring interrupts. When you add an interrupt Component to the design, that interrupt appears in the Interrupts panel of the design-wide resources. In that panel, you identify the core for which the interrupt is enabled and the interrupt priority. For CM0+, you also specify the interrupt vector.

The generated code makes the required calls to the PDL System Interrupt API to configure the interrupts based on your choices. See Configuring Interrupts.

5.3 PSoC Creator and the Base Hardware Address

PSoC Creator generates a symbol for the base hardware address for each Component in the design. The symbol is defined in the header file for the Component. The typical form of this symbol is `<Component_Name>_HW`. For example, if your design has a TCPWM Component named `MyPWM,` the symbol for the base hardware address will be `MyPWM_HW`. You find that symbol in the generated file *MyPWM.h*.

When there are several instances of a peripheral available, the pin assignments you make in the design wide resources file (.cydwr) determine which instance of a peripheral is used. PSoC Creator sets the symbol for the hardware address correctly for that instance.

As a result, you do not need to look in the device-specific header file for the right symbol, or identify which instance of a peripheral is associated with which pin in the hardware design.

You can use the PSoC Creator symbol in any API function call that requires the base hardware address. See also: Finding the Base Hardware Address.

5.4 Configuring a Peripheral with PSoC Creator

PSoC Creator generates a configuration structure for each PDL Component in your top design. You modify options in the Component based on your design requirements.

The configuration structure is named `<ComponentName>_config`. The code is in the files *<ComponentName>.h* and *<ComponentName>.c*. For example, if your design has a TCPWM Component named `MyPWM`, the configuration structure is named `MyPWM_config`. You find it in *MyPWM.h* and *MyPWM.c*.

PSoC Creator automatically generates the code to populate the fields in the configuration structure, based on the choices you make when configuring the component. You can use the structure and symbols defined in these files for any API function call. See Configuring a Peripheral.

In addition, the PSoC Creator Component provides direct access to both the *PDL API Reference* and the Component datasheet. Each is a valuable reference when developing firmware. Simply right-click the Component in the top design.

5.5 Providing Context for a Peripheral with PSoC Creator

PSoC Creator generates the `context` variable for any PDL Component, if one is required.

The context structure is named `<ComponentName>_context`. The code is in the file *<ComponentName>.h.* For example, if your design has an SCB UART Component named `Uart1`, the context structure is declared as a global variable named `Uart1_context` in *Uart1.h*.

```
extern cy_stc_scb_uart_context_t Uart1_context;
```

The context is a PDL-defined but user-allocated data structure. Firmware passes the address of the context structure, but never reads or writes values within the structure. See Providing Context for a Peripheral.

5.6 Using the PSoC Creator Component API

PSoC Creator generates a function API for each Component. You can use the Component API instead of calling PDL functions directly. You can also mix and match these APIs. They are consistent and compatible. Examine the *<ComponentName>.h* file to see what function calls are available to you for any particular Component.

The Component API uses the PDL API. For example, to initialize, enable, and start a TCPWM using the PDL API, your code might look like this (ignoring possible errors):

```
Cy_TCPMW_PWM_Init (TCPWM1, counterNumber, &MyPWM_config);
Cy_TCPWM_Enable_Multiple (TCPWM1, whichCounters);
Cy_TCPWM_TriggerStart (TCPWM1, whichCounters);
```

You properly identify the base hardware address for the TCPWM instance, the counter within that instance, and the address of the configuration structure. The variable `whichCounters` is a bit mask with the bit set for your counter. You must also write the code to declare and fill in the required configuration structure.

By contrast, using the Component API, your code would look like this:

```
MyPWM_Start();
```

PSoC Creator generates the function that handles all the PDL API function calls and parameters.

To set a PWM compare value, a direct call to the PDL API looks like this:

```
Cy_TCPWM_PWM_SetCompare0 (TCPWM1, counterNumber, myValue);
```

The corresponding Component API function call looks like this:

```
MyPWM_SetCompare0 (myValue);
```

The Component API calls the PDL API, but provides the required hardware address and counter number parameters automatically based on the hardware design.

Table 7 summarizes the primary differences between using PSoC Creator generated code, and using the PDL directly.

Table 7. Differences between PSoC Creator and PDL API

Using PSoC Creator	**Using PDL API Directly**
Sets hardware address symbol automatically per peripheral instance	You look up hardware address symbol per peripheral instance
Creates configuration structures	Write your own configuration structures

Unique API per component
(Uses PDL API, automatically provides hardware address)
`MyPWM_SetCompare0()`
`YourPWM_SetCompare()`
	Same function call for any peripheral instance
	(You provide hardware address for the instance)
	`Cy_TCPWM_SetCompare()`
	`Cy_TCPWM_SetCompare()`

May add a layer of indirection	Direct function call
Simplifies the API, handles design-specific parameters automatically	Standard API, you provide all parameters
Generated code can be exported to and used by a third-party IDE	Supports all development models

5.7 Configuring Pins and Trigger Muxing with PSoC Creator

PSoC Creator has a simple user interface to assign pins to particular peripheral inputs and outputs. You make your assignments based upon the design of the target system.

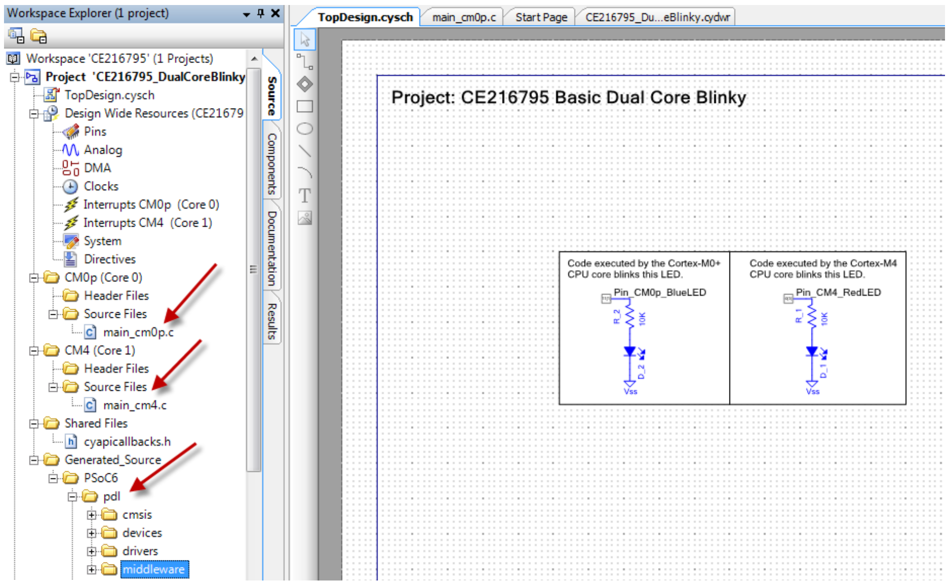
Based on your pin selection, the code generator knows which particular peripheral instance to use, and automatically provides design-based parameters (such as the base hardware address) in PDL API function calls.

In addition, as you wire peripheral inputs and outputs in the PSoC Creator schematic, the code generator handles all required trigger muxing transparently. See Trigger Muxing.

5.8 Developing Dual-Core Applications with PSoC Creator

The PSoC Creator project explorer is organized per core and incorporates everything required by either core. You assign a file to a particular core in that file's properties. Generated source code includes files used by either or both cores. For example, a default PSoC Creator project for a dual-core application has both *main_cm0p.c* and *main_cm4.c*, as shown in Figure 15.

Figure 15. A Dual-Core Project in PSoC Creator



The code also includes the required linker files. When you issue a **Debug** > **Program** command, the executable for each core is loaded into device memory at the appropriate location. The CM0+ core executes first and launches the code on the CM4 core. The default code in *main_cm0p.c* calls `Cy_SysEnableCM4()`.

PSoC Creator supports debugging a single core (either Cortex-M4 or Cortex-M0+) at a time. For more information on debugging PSoC devices with PSoC Creator, refer to the PSoC Creator Help.

5.9 Importing Generated Code into an IDE

A significant number of customers use the PSoC Creator tool as a hardware design platform, but write application software in a different environment. In a typical workflow, a design team creates the design using a PSoC Creator project and generates code. The firmware team can import the generated code into a preferred IDE. As the design evolves, it is simple to update the firmware project with any changes to the design.

There are two ways to import generated code:

- Exporting and importing generated code
- Manually importing generated code

AN219434 – PSoC 6 MCU Importing Generated Code into an IDE covers this topic in detail, including the advantages of each approach.

If you use PSoC Creator to design and configure your system, you gain the significant benefits of generated code. All the necessary configuration code is created for you. This code can be quite complex. For example, the generated code in *cyfitter_cfg.c* initializes all clocks, sets each clock's source and divider, and enables all the clocks in your design. It makes PDL API function calls to do so. This code is based on the clock tree configuration in the PSoC Creator design.

Generated code also provides the Component API. See Using the PSoC Creator Component API.

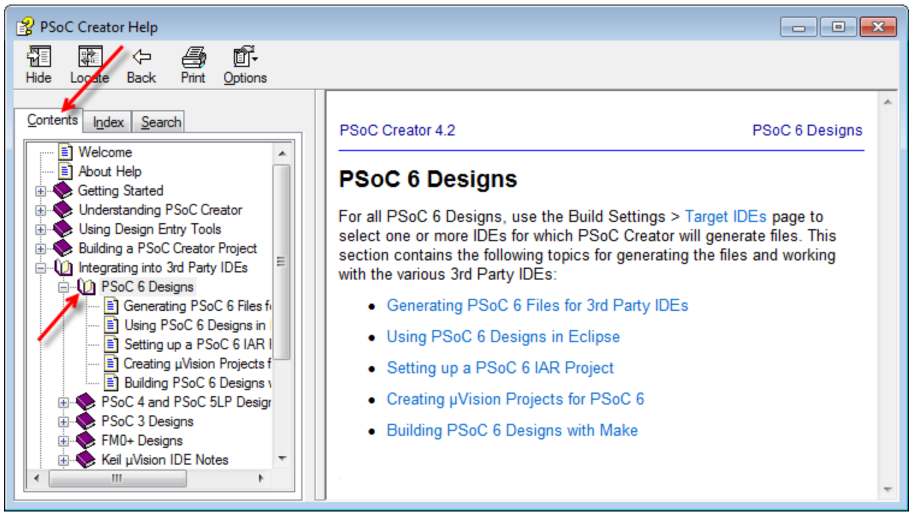
5.9.1 Exporting and Importing Generated Code

For supported IDEs, you can export a package, and then import that package into your IDE.

The export process works the same for all supported IDEs. In the **Project** > **Build Settings** > **Target IDEs** panel you enable export of the generated code to your selected IDE. PSoC Creator generates the export package. The nature and contents of the package varies per IDE.

Import that package into the IDE. Full details about the steps required are in the PSoC Creator help topics (**Integrating 3rd Party IDEs** > **PSoC 6 Designs**).

Figure 16. PSoC Creator Help



PSoC 6 Designs

For all PSoC 6 Designs, use the Build Settings > Target IDEs page to select one or more IDEs for which PSoC Creator will generate files. This section contains the following topics for generating the files and working with the various 3rd Party IDEs:

- Generating PSoC 6 Files for 3rd Party IDEs
- Using PSoC 6 Designs in Eclipse
- Setting up a PSoC 6 IAR Project
- Creating µVision Projects for PSoC 6
- Building PSoC 6 Designs with Make

If you change the design in PSoC Creator, you may need to import a new version of the package. The precise steps required vary per IDE. However, the target IDE recognizes that files have changed.

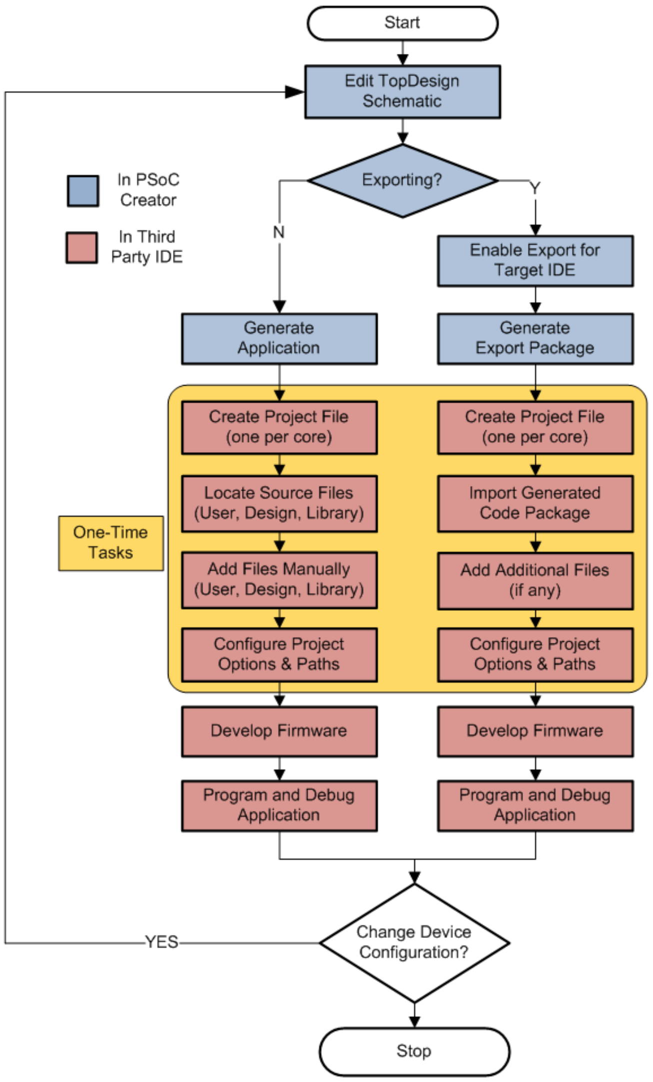
5.9.2 Manually Importing Generated Code

Manual import works for any IDE, including supported IDEs that have an export/import process. For an unsupported IDE, manual import is the only option. To begin, you must have generated code. In PSoC Creator, the **Build** > **Generate Application** command is sufficient. You do not need to compile the code. To manually import generated code, you:

- Create a project in the IDE (one for each core).
- Locate and identify the generated files you need.
- Add those files to the project.

Figure 17 shows how similar the workflow is for each approach.

Figure 17. Integrating Generated Code


Start
Edit TopDesign Schematic
In PSoC Creator
In Third Party IDE
Exporting?
N
Y
Enable Export for Target IDE
Generate Application
Generate Export Package
One-Time Tasks
Create Project File (one per core)
Create Project File (one per core)
Locate Source Files (User, Design, Library)
Import Generated Code Package
Add Files Manually (User, Design, Library)
Add Additional Files (if any)
Configure Project Options & Paths
Configure Project Options & Paths
Develop Firmware
Develop Firmware
Program and Debug Application
Program and Debug Application
YES
Change Device Configuration?
Stop

A. Related Resources

Table A. PDL and PSoC 6 MCU Resources

I Want To	Resources
Evaluate the PDL	• Install PSoC Creator v4.2 and the PDL. • Read this document. • Get the CY8CKIT-062-BLE
Become familiar with the PDL	• Read this document. • Use the PDL API Reference, located in your PDL installation.
Learn About PSoC Creator	• Visit the PSoC Creator web page. • Get the Quick Start Guide. • Get the User Guide. • Watch PSoC 101 video training.
Learn About PSoC 6 MCU	Review available app notes such as: • AN210781 – Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity • AN215656 – PSoC 6 MCU Dual-Core CPU System Design • AN217666 – PSoC 6 MCU Interrupts Review available code examples such as • CE216795 – PSoC 6 MCU Dual-Core MCU Basics • CE212736 – PSoC 6 MCU with Bluetooth Low Energy (BLE) - Find Me • CE213903 – PSoC 6 MCU Basic Bootloaders • CE219881 – PSoC 6 MCU Switching Power Modes • CE218552 – PSoC 6 MCU UART to Memory Buffer Using DMA • CE220541 – PSoC 6 MCU SCB EZI2C
Learn About Custom Components	• Get the PSoC Creator Component Author Guide
Learn About 3rd-Party IDEs and Generated Code	• IAR Embedded Workbench • Keil µVision IDE • GCC ARM Embedded • iSYSTEM winIDEA • Read AN219434 – PSoC 6 MCU Importing Generated Code into an IDE
Become familiar with processor peripherals	• Use the PSoC 6 MCU Technical Reference Manual. • Refer to the datasheet for your device. • Use the PDL API Reference, located in your PDL installation. • Explore the PDL source code.

B. Enabling Debug Probes in µVision IDE

For some platforms (like Windows 10) using a debug probe (such as ULINKpro, ULINK2, J-Link, or CMSIS-DAP) in the µVision IDE requires a custom debug sequence. Without this support, debugging with these probes may fail.

If you use a PDL template project with the µVision IDE, you need to set up support for debug probes. This appendix shows you how.

If you do not use the PDL template projects, you do not need to install the pack described here. If you export code from PSoC Creator for the µVision IDE, the generated pack sets up the debug probe support.

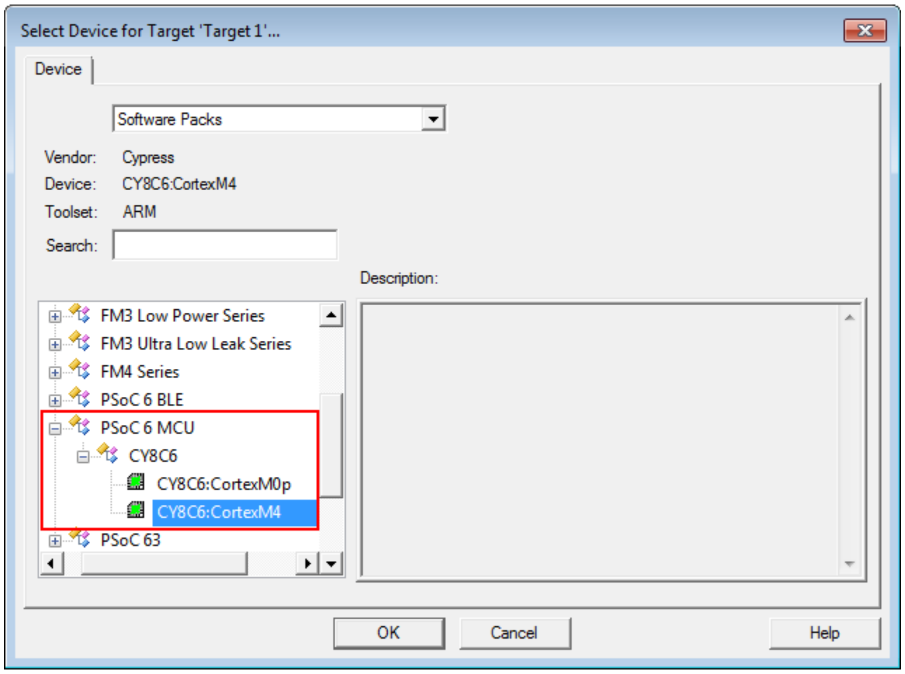
To enable debugging for a PDL template project and the µVision IDE, install the *cypress.uvision_support.1.0.0.pack*. The pack is located here:

<PDL 3.x install folder>\tools\probes\cypress.uvision_support.1.0.0.pack

To install the pack, locate the file and double-click. The Pack Installer installs the necessary files in the µVision IDE.

This is a one-time step that installs support in the IDE. You do not need to repeat this for every project. If you reinstall the µVision IDE, you may need to reinstall this software pack.

A PDL template project uses a generic device. After installing the pack, select the device. Use either CY8C6:CortexM0p or CY8C6:CortexM4 as appropriate.


Select Device for Target 'Target 1'...

Device

Software Packs

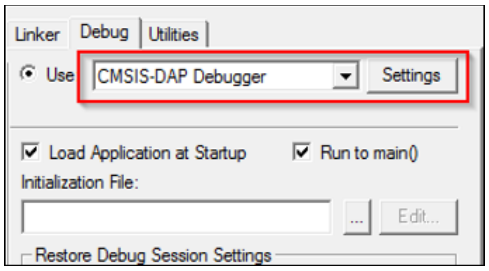
Vendor: Cypress
Device: CY8C6:CortexM4
Toolset: ARM
Search:

Description:

To use a debug probe, do the following.

1. Pick a probe.

In the **Options for Target** window, click the **Debug** tab. Select the appropriate probe and click **Settings**.

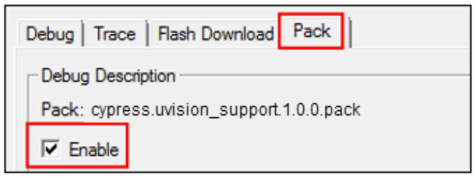


(●) Use: CMSIS-DAP Debugger

✓ Load Application at Startup ✓ Run to main()

2. Check that the pack is enabled.

In the Settings window, click the **Pack** tab, and check that pack support is enabled.

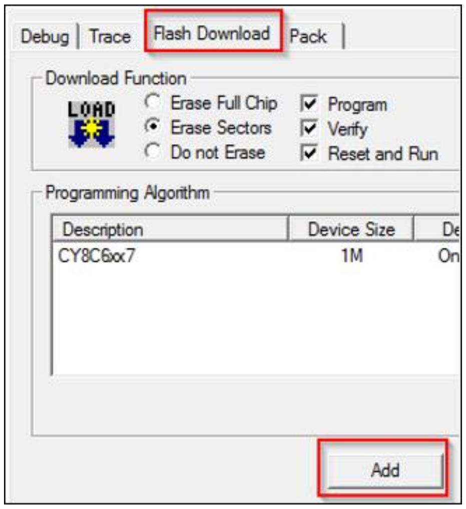


Pack: cypress.uvision_support.1.0.0.pack

✓ Enable

3. Set up the flash loader

Click the **Flash Download** tab, and check that flash loaders are added.



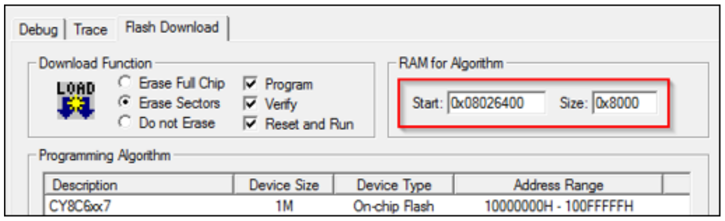
○ Erase Full Chip ✓ Program
● Erase Sectors ✓ Verify
○ Do not Erase ✓ Reset and Run

Description	Device Size	De
CY8C6xx7	1M	On

4. Check the RAM for Algorithm.

For Cortex CM0+ core project: Start: 0x08002400and Size: 0x8000

For Cortex CM4 core project: Start: 0x08026400 and Size: 0x8000



Download Function
- ( ) Erase Full Chip
- (•) Erase Sectors
- ( ) Do not Erase
- ☑ Program
- ☑ Verify
- ☑ Reset and Run

RAM for Algorithm
Start: 0x08026400 Size: 0x8000

Programming Algorithm

Description	Device Size	Device Type	Address Range
CY8C6xx7	1M	On-chip Flash	10000000H - 100FFFFFH

5. **Save your changes.**

   Click **OK** to exit the **Settings** and the **Target for Options** windows.

You can now program the target and debug the project.

NOTE: In some cases, after flashing you will need to reset the device to run the programmed application.

Revision History

Document Revision History

Document Title: Peripheral Driver Library v3.0 User Guide Document Number: 002-18032			
Revision	Issue Date	Origin of Change	Description of Change
**	12/26/2016	JETT	Initial release
*A	03/24/2017	JETT	Changed Title from "Quick Start Guide" to "User Guide" Updated screenshots to maintain accuracy Updated Section 3.1 to use a dual-core code example Expanded Section 4 to cover more topics of interest Limited Section 4 to IDE-agnostic information Added Section 5 to expand on the PSoC Creator development model Minor grammar and wording improvements throughout Updated template
*B	07/14/2017	JETT	Updates for release 3.0.1, including new drivers and PSoC Creator 4.2 Section 2.1 table 1 Crypto driver is delivered as binary libraries; added analog drivers Section 2.3 added information about global search in the PDL API reference Section 3.2 added troubleshooting information on the debugger connection Section 4.6 and 5.5 refined discussion of how to use a context for a peripheral Section 4.11 removed the combined Port/Num macros discussion Section 5.9 updated to refer to the new AN on importing generated code Minor grammar improvements throughout
*C	08/04/2017	JETT	Removed NDA limitation from footer. Eliminated unnecessary cross references and hyperlinks to local images and tables. Section 4.3 – noted that some CM0+ IRQs are reserved for system use, refer to the API documentation for details.
*D	08/31/2017	JETT	Changed a path reference from 3.0.0 > 3.0.x Updated App Note and Code Example titles Added guidance on dual-core and single-core debugging with third-party IDEs
*E	10/27/2017	JETT	Updated Table 1 the list of supported drivers Updated Table 2 for new em-EEPROM middleware Updated Figure 8 for a UI change Fixed minor grammar/spelling issues
*F	12/05/2017	JETT	Table 2: update information on security template Section 3.2: update troubleshooting information for change in KitProg behavior Section 4.11: new section on power mode transitions Section 4.15: update name and description of CyMCUElfTool Section 4.16: new section on secure systems
*G	12/13/2017	JETT	Update IDEs listed in Related Resources

Revision	Issue Date	Origin of Change	Description of Change
*H	01/29/2018	JETT	Section 4.15: Update information on single-core device initialization Section 4.17: Add information on ULINKPro and CMSIS DAP debug probe support; update figure 14 for list of supported IDEs Update title of AN 219434 Minor tweaks to link formatting Update copyright dates
*I	2/13/2018	JETT	Added Appendix B on debug probe support for µVision Updated figure 4 Fixed typos related to System Power Management
*J	4/12/2018	JETT	Updated template and logo images to fix dithering at high magnification Sections 2 and 3: added “where to get” information; the latest version may not be installed by PSoC Creator Updated Figures 2, 3, and 5 for version number, and (figure 3) for minor UI changes in the PDL documentation