



# Cypress Peripheral Driver Library v2.1 Quick Start Guide

Doc. No. 002-13955 Rev \*\*

Cypress Semiconductor  
198 Champion Court  
San Jose, CA 95134-1709  
Phone (USA): 800.858.1810  
Phone (Intl): +1 408.943.2600  
[www.cypress.com](http://www.cypress.com)

© Cypress Semiconductor Corporation, 2016. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you under its copyright rights in the Software, a personal, non-exclusive, nontransferable license (without the right to sublicense) (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units. Cypress also grants you a personal, non-exclusive, nontransferable, license (without the right to sublicense) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely to the minimum extent that is necessary for you to exercise your rights under the copyright license granted in the previous sentence. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and Company shall and hereby does release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. Company shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit [cypress.com](http://cypress.com). Other names and brands may be claimed as property of their respective owners.

# Contents



<b>1. Introduction.....</b>	<b>4</b>
<b>2. Peripheral Driver Library Overview.....</b>	<b>5</b>
2.1 Changes from PDL v2.0 .....	6
2.2 Getting and Installing the PDL .....	6
2.3 PDL Organization .....	7
2.4 Using PDL Code Examples .....	7
<b>3. Build and Run a PDL Project.....</b>	<b>8</b>
3.1 Before You Begin .....	8
3.2 Building with IAR Embedded Workbench .....	11
3.3 Building with Keil $\mu$ Vision.....	13
3.4 Building with iSYSTEM WinIDEA .....	14
3.5 Building with Atollic TrueSTUDIO .....	16
3.6 Building with GCC ARM Embedded Tools .....	19
3.7 Troubleshooting.....	21
<b>4. . Developing Code Using the PDL.....</b>	<b>22</b>
4.1 Creating a Custom Project .....	22
4.2 Configuring the PDL .....	23
4.3 Configuring a Peripheral.....	24
4.4 Using a Peripheral .....	25
4.5 Working with Registers.....	26
4.6 Managing Pins.....	27
4.7 Using the PDL to Learn Low-level Programming.....	28
4.8 PDL Documentation .....	29
4.9 Naming Conventions .....	30
<b>A. PDL-Related Resources.....</b>	<b>31</b>
<b>Revision History.....</b>	<b>32</b>
Document Revision History .....	32

# 1. Introduction



Cypress provides the Peripheral Driver Library (PDL) v2.1 to simplify software development for the FM0+ and FM4 MCU portfolios. PDL v2.0 also supports the FM3 portfolio.

You configure the library for the desired functionality. You then use API calls to initialize and use a peripheral. The design of the PDL enables custom driver development. Using the PDL also makes it easier to port code from one portfolio to the other, because the same code supports multiple FM portfolios.

The PDL reduces the need to understand register usage and bit structures, thus easing software development for the extensive set of peripherals in the FM portfolios. 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 modify register values directly in your code. The PDL makes this unnecessary.

However, developers who wish to work at the register level can use the PDL source code as a guide. Combined with study of the appropriate data sheet and peripheral manual, you can learn the information you need to create your own driver and use a peripheral.

This document details the features and capabilities of PDL v 2.1. Key changes from version 2.0 are also discussed.

If you work with PDL v2.0, use the Quick Start Guide for that version.

## 2. Peripheral Driver Library Overview



The PDL provides a high-level API to configure, initialize, and use a peripheral driver. The PDL also provides numerous code examples that demonstrate how to use the peripherals. The PDL includes the necessary startup code for each supported device. PDL v2.1 supports all devices in the FM0+ and FM4 portfolios. For the FM3 portfolio, please use PDL v2.0.

Table 2-1 lists the peripherals supported by the PDL.

Table 2-1. Peripheral Support in the PDL

Peripheral	Description	Peripheral	Description
ADC	Analog Digital Converter	I2S	Inter IC Sound
BT	Base Timer	ICC	IC Card Interface
CAN	Controller Area Network	LCD	Liquid Crystal Display
CLK	Clock Functions	LPM	Low Power Modes
CR	RC Oscillator trimming	LVD	Low Voltage Detection
CRC	Cyclic Redundancy Check	MFS	Multi-Function Serial Interface
CSV	Clock Supervisor	MFT	Multi-Function Timer
DAC	Digital Analog Converter	PCRC	Programmable CRC
DMA	Direct Memory Access	PPG	Programmable Pulse Generator
DSTC	Descriptor System Data Transfer Controller	QPRC	Quadrature Position/Revolution Counter
DT	Dual Timer	RC	Remote Control (HDMI-CEC/Remote Control Reception/Transmission)
EXINT	External Interrupts	RESET	Reset
EXTIF	External Bus Interface	RTC	Real Time Clock
Flash	Flash Memory	SDIF	SD Card Interface
GPIO	General Purpose I/O Ports	UID	Unique ID
HBIF	HyperBus Interface	VBAT	VBAT Domain
HSSPI	High Speed Serial Peripheral Interface	WC	Watch Counter
I2CS	Inter IC Slave	WDG	Software and Hardware Watchdog Counter

The PDL is a superset of all the code required to build any driver for any supported device. This superset design means:

- All APIs needed to initialize, configure, and use a peripheral are available.
- The PDL includes error checking to ensure the targeted peripheral is present on the selected device.

The superset design means the PDL is useful across all devices, whatever peripherals are available. This enables code to maintain compatibility across platforms where peripherals remain present. If you configure the PDL to include a peripheral that is unavailable on the hardware, your project will fail at compile time, rather than at runtime. The PDL configuration logic knows the target microcontroller and removes the peripheral register headers for unsupported peripherals from the build.

**Before writing code to use a peripheral, consult the datasheet for the particular series or device to confirm support for the peripheral.**

## 2.1 Changes from PDL v2.0

Table 2.2 lists the major changes and improvements:

Table 2-2. PDL Changes

Item	PDL v2.0	PDL v2.1
<b>Portfolio Support</b>	FM0+, FM3, FM4	FM0+, FM4
<b>Peripherals</b>		Removed I2SL (the same functionality is in I2S)
<b>Code Examples</b>	A single project file for each IDE, must reconfigure for every microcontroller	A project file for each code example, for each IDE
<b>Low-level programming</b>		Preconfigured project file provided for each device package, for each IDE (does not include PDL files)
<b>IDE Support</b>	IAR Embedded Workbench 7.5 Keil µVision v 4.7	IAR Embedded Workbench 7.50.3 Keil µVision v 5.18 iSYSTEM winIDEA v9.12 Atollic TrueSTUDIO v5.5.2 GCC makefile
<b>Documentation</b>		Includes overview and setup information for each peripheral Enhanced the left menu navigation to provide direct access to #defines, enums, structures, and functions
<b>Device support</b>	Monolithic, used <code>#ifdef</code> to configure PDL for a particular series and package	Dedicated device header file and startup code for each device package, using CMSIS v4.5
<b>pdl_device.h</b>	Required to specify series and package for gpio.h	No longer used

Note that PDL v2.1 does not currently support the FM3 portfolio. Developers targeting the FM3 portfolio should continue to use PDL 2.0.x.

## 2.2 Getting and Installing the PDL

Most Cypress FM0+ and FM4 starter kits install the PDL. See the [PDL-Related Resources](#) section for links to choices. A kit may install an older version of the PDL because it requires that older version.

If your kit does not install the PDL, or you want the latest version, you can download the PDL Installer from the [Cypress PDL](#) product page.

The [Build and Run a PDL Project](#) section shows a step-by-step process to build and debug a PDL project, using a PDL code example.

The [Developing Code Using the PDL](#) section provides a basic introduction to programming with the PDL.

## 2.3 PDL Organization

The PDL is organized into several folders. [Table 2-3](#) shows the PDL folder structure.

Table 2-3. PDL Folder Structure

Path\Folder	Description
<i>cmsis</i>	Header files from CMSIS spec v4.5
<i>devices</i>	For each device package: <ul style="list-style-type: none"> <li>• common header files</li> <li>• configuration, startup, and project files for each IDE</li> </ul>
<i>doc</i>	PDL documentation
<i>driver</i>	Driver source code and header files
<i>examples</i>	Code examples for each peripheral on each supported starter kit
<i>utilities</i>	Various utility files

When you use the PDL, typically the only files you modify are *pdl\_user.h* and *main.c*.

## 2.4 Using PDL Code Examples

The PDL installation includes code examples for particular starter kits. You find the code examples in the *examples* folder.

Each example demonstrates the basic initialization and configuration for a peripheral. Some peripherals have multiple examples.

To use a code example, just double click the project file for your preferred IDE. The [Build and Run a PDL Project](#) section shows a step-by-step process to build a PDL project, using a PDL code example.

The [Developing Code Using the PDL](#) section provides a basic introduction to programming with the PDL.

## 3. Build and Run a PDL Project



In this section you build and run an example PDL project. The goal is to ensure that you can use the PDL in a supported IDE. We want to make sure you have everything installed and can use it. These instructions do not teach you how to use the PDL.

See [Developing Code Using the PDL](#) for information on how to create a custom project or configure and use the PDL. In particular, see the [PDL Documentation](#) section for information on how to use the technical documentation for the PDL and learn about the PDL's function API.

Each code example includes a project file for each supported IDE. These project files are configured for the particular IDE, example, and board.

### 3.1 Before You Begin

Ensure that you have three items:

- Hardware on which to run the code
- The PDL
- A development environment.

Then connect your board to your computer.

#### Hardware

The PDLv2.1 supports FM0+ and FM4 portfolio microcontrollers. These instructions use either the [FM4 S6E2GM-Series MCU Pioneer Kit](#) shown in [Figure 3-1](#), or the [FM0+ S6E1B8-Series Starter Kit](#) shown in [Figure 3-2](#).

Modify the details in these instructions to adapt to other kits, or your own hardware.

#### PDL v2.1

If you do not have the PDL installed, see [Getting and Installing the PDL](#).

#### Development Environment

Ensure that you have installed a development environment. PDL v2.1 provides project files for:

- [IAR Embedded Workbench v7.50.3](#)
- [Keil μVision IDE v5.18](#)
- [iSYSTEM winIDEA v9.12](#)
- [Atollic TrueSTUDIO v5.5.2](#)
- [GNU ARM Embedded tools](#)

#### Connect your board to your computer

The precise details will vary based on your hardware. Each of the kits described has two connectors. For the S6E2GM starter kit, use CN2 ([Figure 3-1](#)). For the S6E1B8 starter kit, use CN3 ([Figure 3-2](#)). These are the connectors near the corner of the board. Use the provided cable and connect your computer to the board.

When properly connected, **LED3 Power** on the board will be green.



In the following sections (one per IDE), we use the *bt\_pwm* code example to walk you through the process of opening a project file, building the code, and downloading to a debugger. This example uses the PDL to initialize and use the Base Timer (BT) peripheral to generate a PWM. The code uses the PWM to blink the RGB LED on the board.

This example works on either the FM4 S6E2GM, or the FM0+ S6E1B8 kit. The only difference is the path to the project file.

To learn how the code example enables the PDL timer functionality, review the source code or see [Configuring the PDL](#).

If you run into problems, see the [Troubleshooting](#) section for some of the more likely causes and solutions.

Figure 3-1. The S6E2GM-Series Pioneer Kit Board

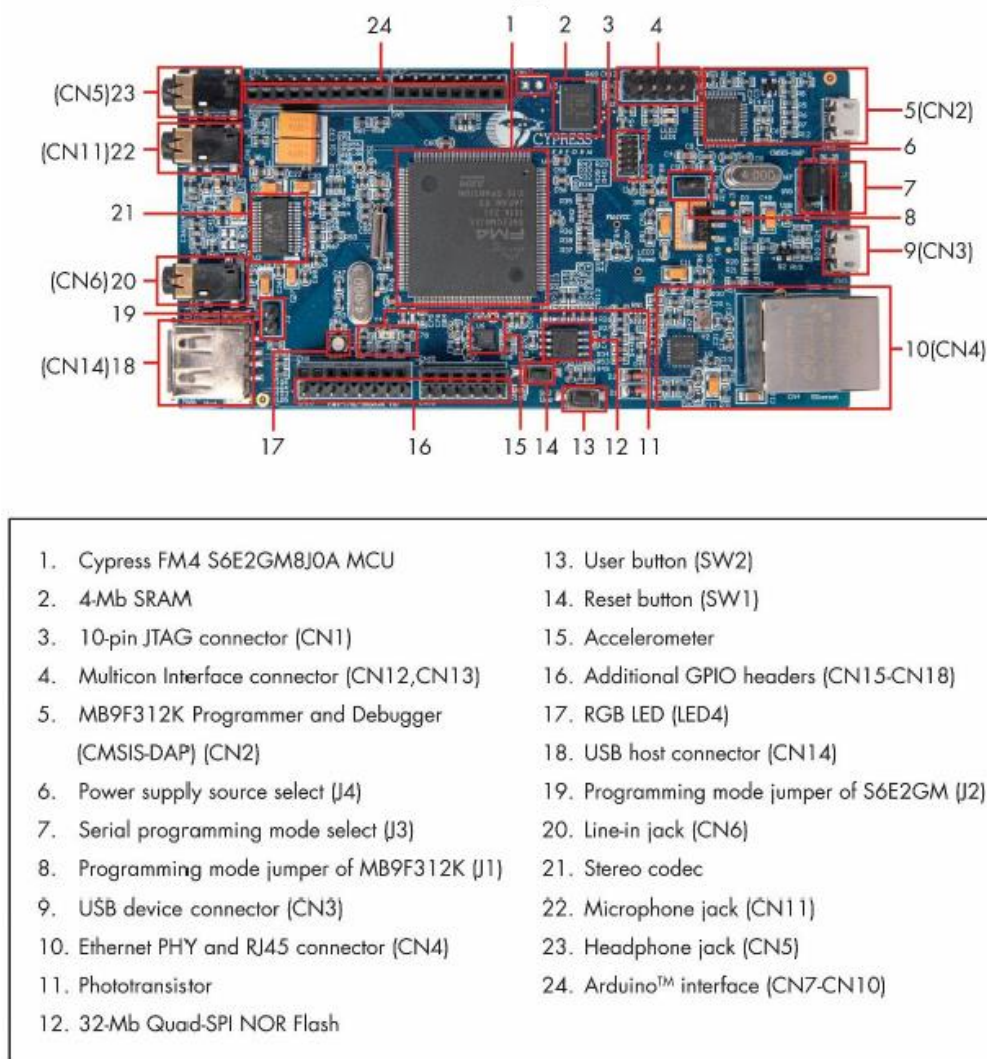
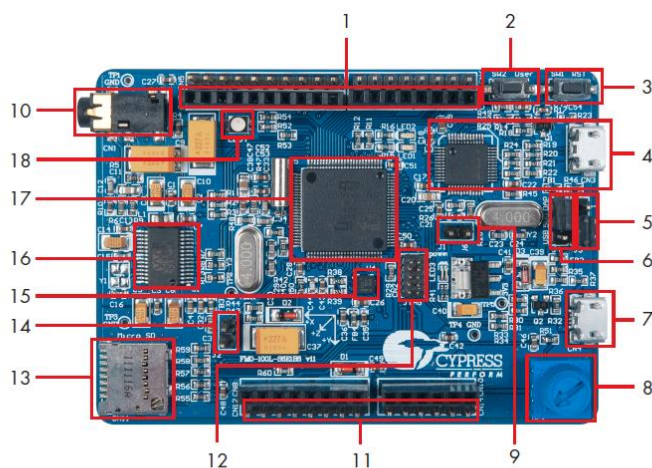


Figure 3-2. The S6E1B8-series Starter Kit Board



- |  |  |
|--|--|
| 1. Arduino interface (CN7-CN10)              | 11. Additional GPIO headers (CN5, CN6, CN12, CN14) |
| 2. User button                               | 12. 10-pin JTAG connector (CN2)                    |
| 3. Reset button                              | 13. Micro SD card connector (CN11)                 |
| 4. Programmer and debugger (CMSIS-DAP)       | 14. Programming mode jumper of FMO+ (J2)           |
| 5. Series programming mode select (J3)       | 15. Accelerometer                                  |
| 6. Power supply select (J4)                  | 16. Stereo codec                                   |
| 7. USB device connector (CN4)                | 17. Cypress FMO+ MCU S6E1B86FOA                    |
| 8. Potentiometer                             | 18. RGB LED  |
| 9. Programming mode jumper of MB9AF312K (J1) |  |
| 10. Headphone and microphone jack (CN1)      |  |

## 3.2 Building with IAR Embedded Workbench

These instructions assume you are familiar with the IAR tools.

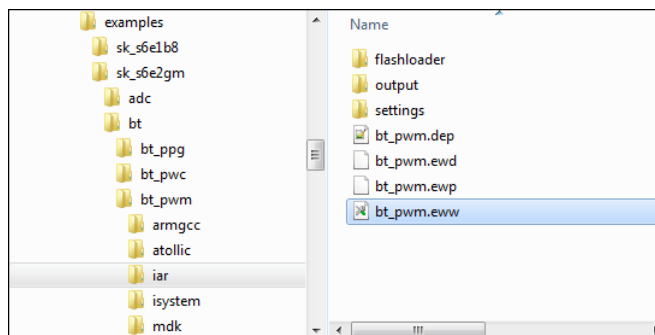
### 1. Open the workspace file.

For the S6E2GM kit, the path to the file is: `<PDL directory>\examples\sk_s6e2gm\bt\bt_pwm\iar\`

For the S6E1B8 kit, the path to the file is: `<PDL directory>\examples\sk_s6e1b8\bt\bt_pwm\iar\`

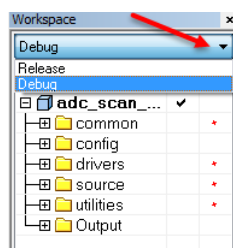
Double click the IAR workspace file: `bt_pwm.eww`. The project file opens.

Figure 3-3. Open the Workspace File



Make sure the **Debug** build is selected. This build is configured to generate debug symbols. Change it if necessary.

Figure 3-4. Use the Debug Target

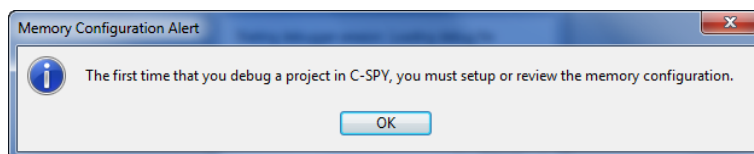


### 2. Download the code and launch the debugger.

Choose **Project > Download & Debug**. The IDE compiles, links, downloads the code to the board, and launches the debugger.

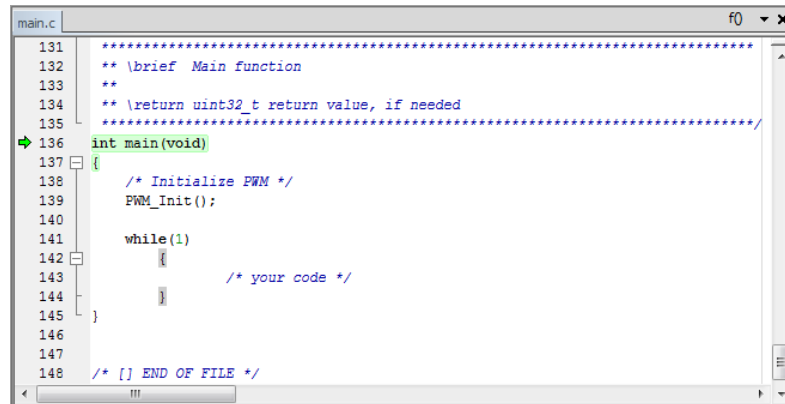
You may see an alert as shown in Figure 3-5. If you do, click **OK** to see the memory configuration. Click **OK** again to accept the memory configuration.

Figure 3-5. Memory Configuration Alert



When the download is complete, the program counter halts at the first line of *main()*.

Figure 3-6. Debugger Halted at the *main()* Function



### 3. Run the code and observe the results.

In the IAR tools, choose **Debug > Go**. The RGB LED blinks slowly.

Congratulations, you have successfully built and executed a simple PDL-based embedded application.

### 3.3 Building with Keil $\mu$ Vision

These instructions assume you are familiar with the  $\mu$ Vision tools.

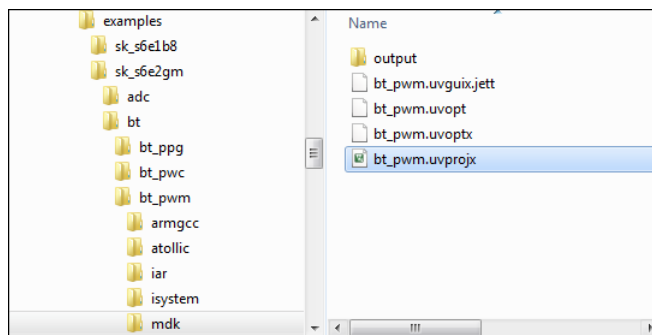
#### 1. Open the project file.

For the S6E2GM kit, the path to the file is: <PDL directory>\examples\sk\_s6e2gm\bt\bt\_pwm\mdk\

For the S6E1B8 kit, the path to the file is: <PDL directory>\examples\sk\_s6e1b8\bt\bt\_pwm\mdk\

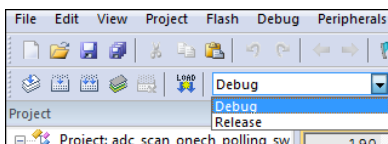
Double click the  $\mu$ Vision project file: *bt\_pwm.uvprojx*. The project file opens.

Figure 3-7. Open the Project File



Make sure the **Debug** build is selected. This build is configured to generate debug symbols. Change it if necessary.

Figure 3-8. Use the Debug Target



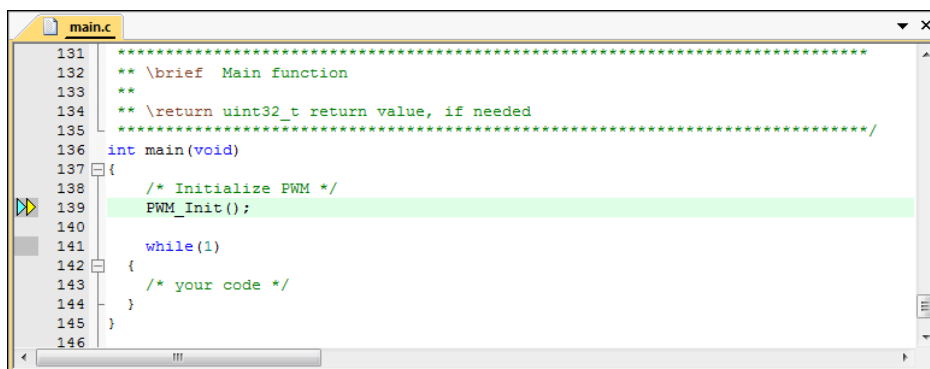
#### 2. Compile the code.

Choose **Project > Build Target**. The IDE compiles and links the code.

#### 3. Download the code and launch the debugger.

Choose **Debug > Start/Stop Debug Session**. The IDE downloads the code and launches the debugger. When the download is complete, the program counter halts at the first line of *main()*.

Figure 3-9. Debugger Halted at the *main()* Function



#### 4. Run the code and observe the results.

In the  $\mu$ Vision tools, choose **Debug > Run**. The RGB LED blinks slowly.

Congratulations, you have successfully built and executed a simple PDL-based embedded application.

## 3.4 Building with iSYSTEM WinIDEA

These instructions assume you are familiar with the WinIDEA tools.

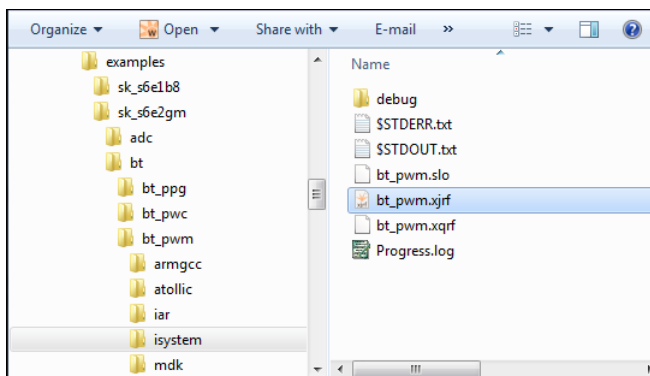
### 1. Open the workspace file.

For the S6E2GM kit, the path to the file is: `<PDL directory>\examples\sk_s6e2gm\bt\bt_pwm\isystem\`

For the S6E1B8 kit, the path to the file is: `<PDL directory>\examples\sk_s6e1b8\bt\bt_pwm\isystem\`

Double click the iSYSTEM workspace file: `bt_pwm.xjrf`. The project file opens.

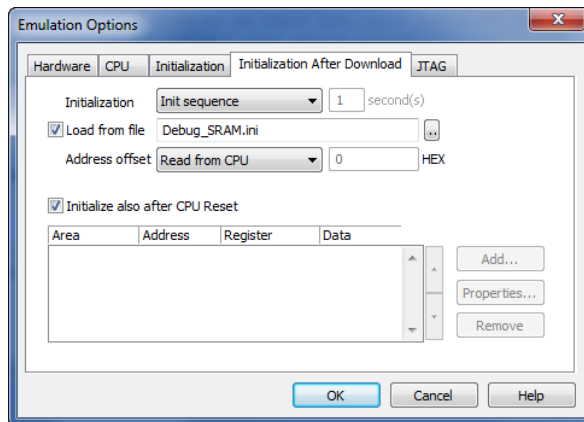
Figure 3-10. Open the Workspace File



Make sure the **Debug** build is selected. This build is configured to generate debug symbols. Use **Project > Targets** to change it, if necessary.

**Note:** Because the **Debug** build downloads to RAM, the project uses an initialization script to set up the Vector Table Offset Register (VTOR), program counter, and stack pointer. You don't need to do anything if you use the Debug build, this is set up for you. However, to run the **Release** build (which downloads to Flash), you must set **Initialization** to **none**. Use **Hardware > Emulation Options > Initialization After Download** to see this setting.

Figure 3-11. Initialization Options Set for the Debug Build



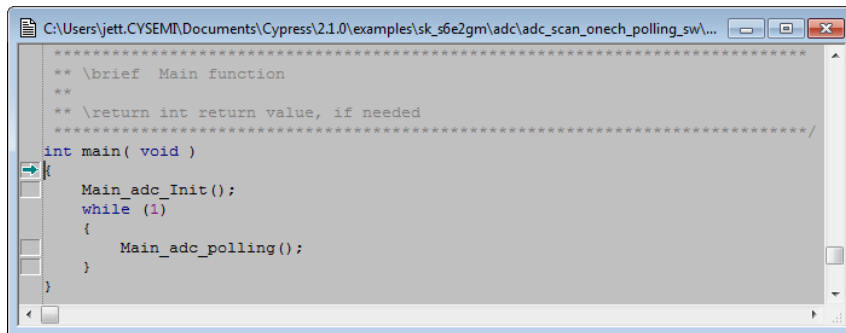
**2. Compile the code.**

Chose **Project > Make** to build the application. The code should compile with no warnings or errors.

**3. Download the code and launch the debugger.**

Choose **Debug > Download**. The IDE downloads the code and launches the debugger. When the download is complete, the program counter halts at the first line of *main()*.

Figure 3-12. Debugger Halted at the *main()* Function

**4. Run the code and observe the results.**

In the WinIDEA tools, choose **Debug > Run Control > Run**. The RGB LED blinks slowly.

Congratulations, you have successfully built and executed a simple PDL-based embedded application.

## 3.5 Building with Atollic TrueSTUDIO

These instructions assume you are familiar with the TrueSTUDIO tools.

Atollic tools rely on third-party debuggers. Support for FM portfolio microcontrollers varies among these tools. For example, the Segger J-LINK does not currently support the S6E2GM microcontroller. These steps provide instructions if this known issue affects you.

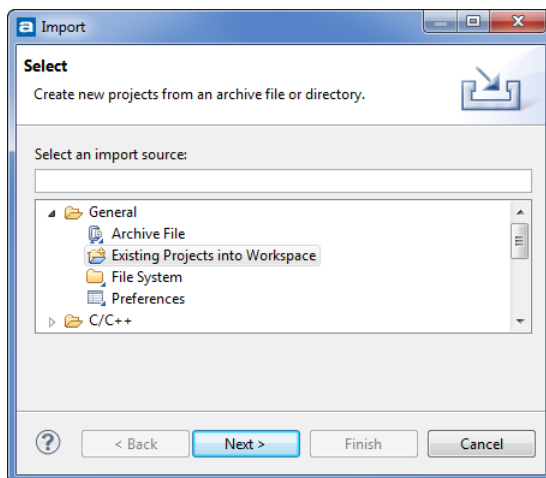
### 1. Import the project file to your workspace.

- A. Launch TrueSTUDIO and choose the workspace you wish to use.

Use **File > Import** to open the Import dialog. Expand **General**, select **Existing Projects into Workspace**.

Then click **Next**.

Figure 3-13. Importing the Project File



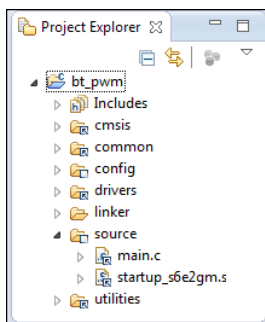
- B. Browse to locate the directory that contains the project file.

For the S6E2GM kit, the path to the file is: `<PDL directory>\examples\sk_s6e2gm\bt\bt_pwm\atollic\`

For the S6E1B8 kit, the path to the file is: `<PDL directory>\examples\sk_s6e1b8\bt\bt_pwm\atollic\`

Click **Finish**. The project file opens.

Figure 3-14. Code Example Project in TrueStudio



- C. Make sure the **Debug** build is active.

The Debug build is configured to generate debug symbols. Use **Project > Manage Build Configurations** to change it if necessary.



## 2. Compile the code.

Chose **Project > Build Project** to build the application. The code should compile with no warnings or errors.

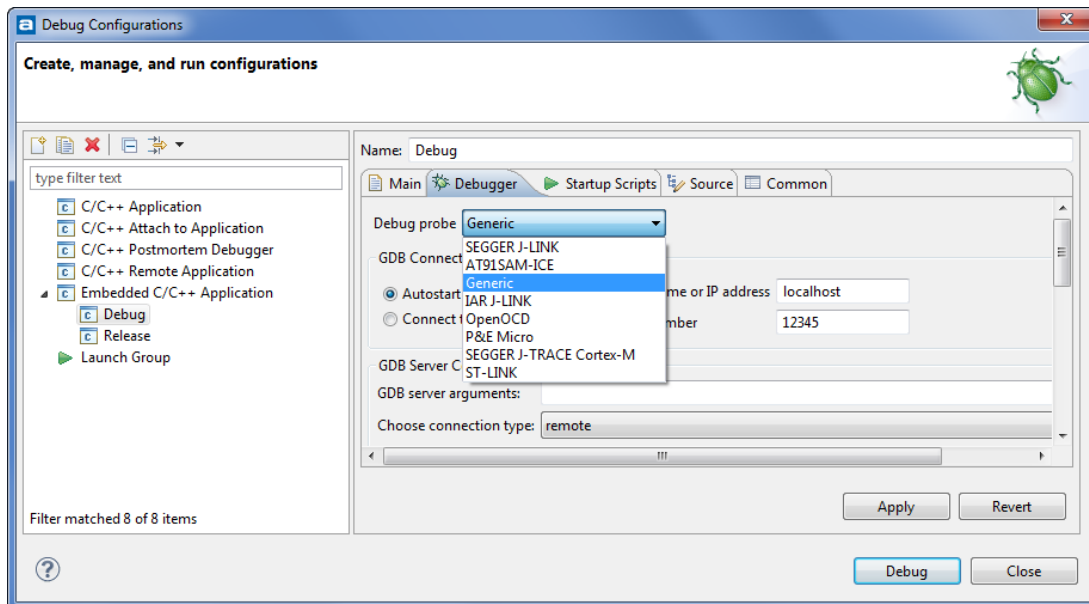
## 3. Choose a debugger.

Some debuggers, such as SEGGER J-LINK, do not support programming the latest FM portfolio microcontrollers. In that case you need to program the flash using a different tool. If this affects you, see Step 5 below.

If your debugger supports your FM microcontroller, then choose **Run > Debug Configurations**. Expand the Embedded C/C++ Application category, and select the **Debug** build.

Click the **Debugger** tab and change the configuration to choose the debugger you prefer. Set options as required.

Figure 3-15. Choosing a Debugger



Then click **Debug**. The IDE downloads the code and launches the debugger. When the download is complete, the program counter halts at the first line of `main()`.

## 4. Run the code and observe the results.

In the TrueSTUDIO tools, click the **Resume** button. The RGB LED blinks slowly.

Figure 3-16. Click the Resume Button to Run Code



## 5. Download the executable to the board manually.

If your debugger choice in TrueSTUDIO does not support flash programming for your FM microcontroller, you can download the executable to the board manually using a stand-alone flash programmer.

Cypress provides stand-alone flash programmer tools for FM portfolio devices:

- [FLASH MCU Programmer for FM0+/FM3/FM4](#)
- [FLASH USB Direct Programmer](#)

Full instructions on how to use either tool are in:

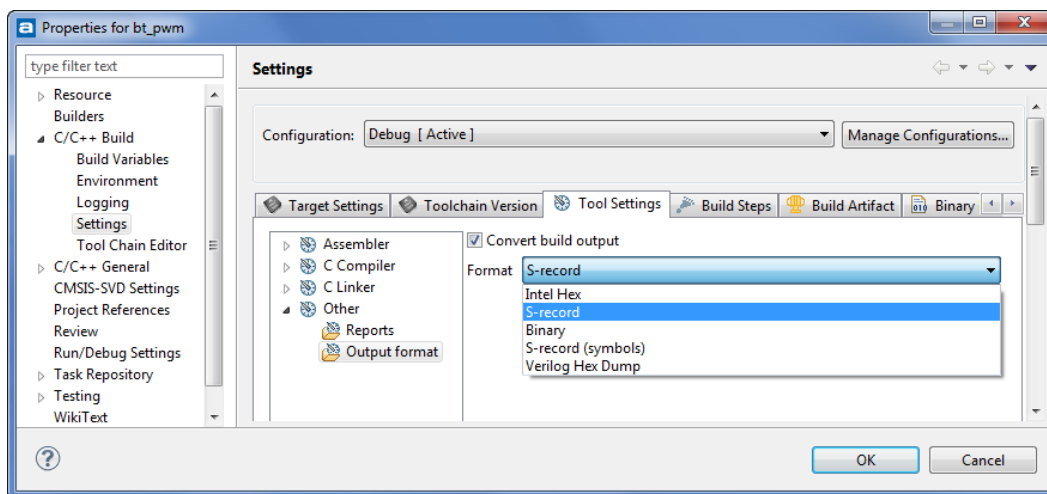
- [AN211122: Getting Started with FM4 Development](#)
- [AN210985: Getting Started with FM0+ Development](#)

These tools support executables in either Motorola S-Record or Intel-HEX formats. You can configure the TrueSTUDIO tools to produce the executable in either format. Navigate to:

**Project > Build Settings > C/C++ Build > Settings > Tool Settings > Other > Output Format.**

Enable the **Convert build output** option, and specify the output format you want to generate.

Figure 3-17. Specify Output Format



## 3.6 Building with GCC ARM Embedded Tools

These instructions assume you are familiar with the GCC ARM Embedded tools.

You can use GCC ARM Embedded tools with a variety of operating systems, and a variety of third-party tools. These instructions use the Windows operating system, and the Minimalist GNU for Windows ([MinGW](#)) tools.

### 1. Set up an environment variable.

The make file requires an environment variable named ARMGCC\_DIR.

```
TOOLS_DIR = "$ (ARMGCC_DIR) "
```

This environment variable must contain the path to the GCC Arm Embedded tools. To create the environment variable:

#### A. Open the Environment Variables window.

Right-click **Computer**, choose **Properties**.

Click **Advanced system properties**.

Click the **Environment Variables** button.

The Environment Variables dialog window opens.

#### B. Create a new environment variable.

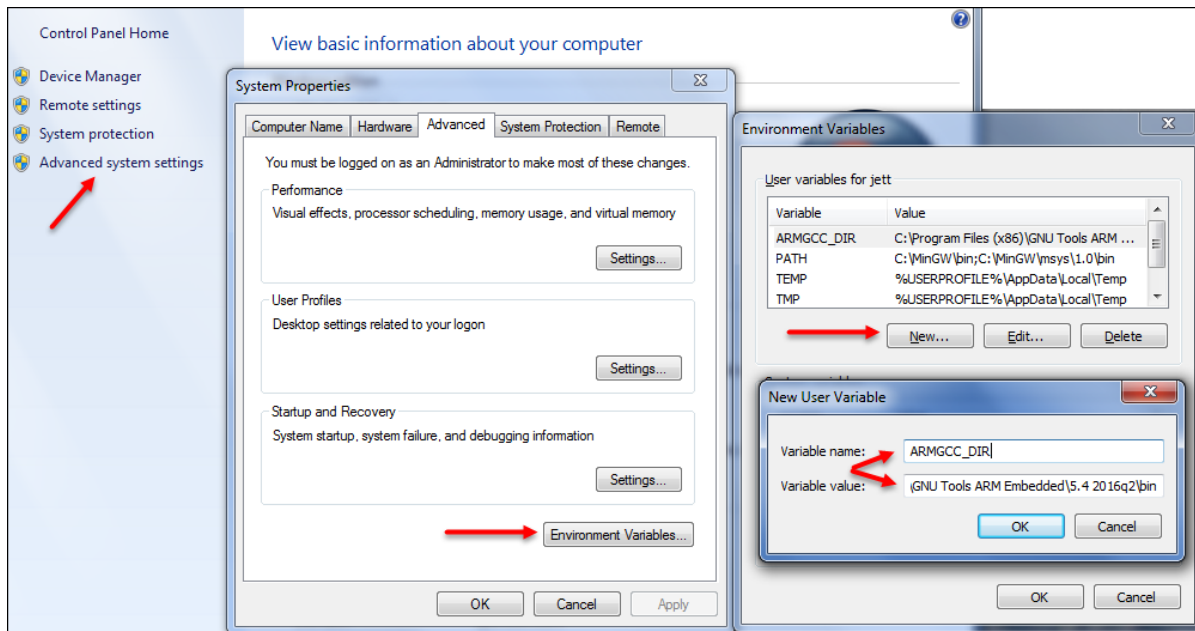
Click the **New** button.

Set the **Variable name** to ARMGCC\_DIR. Set the **Variable value** to the path to your GCC ARM Embedded tools. For example, it might be *C:\Program Files (x86)\GNU Tools ARM Embedded\5.4 2016q2\bin*.

#### C. Close the dialog windows.

Click **OK** as required to close the nested dialog windows.

Figure 3-18. Create an Environment Variable



## 2. Run the make.bat file for the code example.

For the S6E2GM kit, the path to the file is: `<PDL directory>\examples\sk_s6e2gm\bt\bt_pwm\armgcc\`

For the S6E1B8 kit, the path to the file is: `<PDL directory>\examples\sk_s6e1b8\bt\bt_pwm\armgcc\`

Double click the file: *make.bat*.

The command line tools open, compile, and link the executable. The make file is set up to save the executable in both *elf* and *srec* formats.

## 3. Download the executable to the board.

Use a flash programmer to download the executable to the board. The details of which file format you use, how to connect to the board, and how to flash the board will depend upon your tools and preferences.

Cypress provides stand-alone flash programmer tools for FM portfolio devices. These tools support executables in either Motorola S-Record or Intel-HEX format.

- [FLASH MCU Programmer for FM0+/FM3/FM4](#)
- [FLASH USB Direct Programmer](#)

Full instructions on how to use either tool are in:

- [AN211122](#): Getting Started with FM4 Development
- [AN210985](#): Getting Started with FM0+ Development

## 4. Run the executable.

The make file is configured to generate debug symbols

```
COMPILER_OPTIONS = -g -ggdb -Os -Wall -fno-strict-aliasing
```

If you wish to debug the code, use your preferred debugger to connect to the executable on the board. Alternatively, use your debugger to flash the board with the *elf* or *srec* file, and start execution.

## 3.7 Troubleshooting

You may run into an issue as you attempt to build and run a code example. This section discusses the more likely scenarios, and what you can do about the problem. We assume that you are familiar with your particular build environment, and that all paths and environment variables required by the tools are known to be good.

### Project doesn't open

- Make sure you are using a supported version of the tools.
- Launch the IDE first, and then try to open the project file or workspace.

### Board isn't getting power

- Ensure that the USB cable is attached to the right connector on your board.
- Check jumper connections on the board.
- Try a different USB cable. They do fail sometimes.

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

- Ensure that the USB cable is attached to the right connector on your board.
- Check jumper connections on the board.
- Make sure 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 the Device Manager to ensure there is a COM port for the board.
- If you are using a different debugger than the default settings in the code example project, ensure that your debug options are set correctly for your connection.
- If all else fails, reboot to clear any COM port conflicts that might occur in the Windows OS

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

- Make sure that you are using the **Debug** build. (The **Release** build typically does not generate debug symbols, and has optimizations turned on, which can interfere with debugging line-by-line.)
- Ensure that your environment's debug options are set to run to the first line of *main()*.
- Ensure that the debug options in your IDE for your debugger are correct. The code example projects use CMSIS-DAP when that is supported by the IDE.

### Error when compiling (e.g. file not found).

IDE projects maintain hard-coded paths to source files and depend on the PDL's default folder structure. To avoid problems, the PDL is entirely self contained, meaning the code examples are inside the PDL installation. If you relocate files, or try to move a Starter Kit code example into the PDL, the projects will break.

If you have relocated files or folders in the PDL installation:

- Restore the files and folders to their original locations.
- Reinstall the PDL, which restores the original folder structure.

To use a Starter Kit code example:

- You must use it inside the Starter Kit itself. (Like the PDL, the Starter Kit is self-contained. Each kit installs its own copy of the PDL, in a known location.)
- Double-click the project file inside the Starter Kit code examples, in the Starter Kit installation.

## 4. . Developing Code Using the PDL

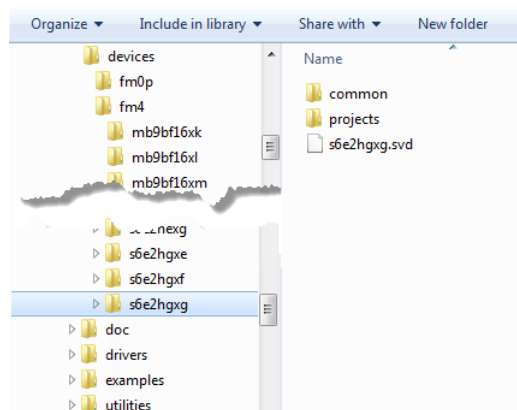


### 4.1 Creating a Custom Project

A PDL code example project is configured for the microcontroller found in a starter kit. You may want to create a project that targets a different microcontroller.

Device-specific files are in the `<PDL directory>\devices\` folder. Navigate to locate the device you want, as shown in Figure 4-1, which uses S6E2HGxG as an example.

Figure 4-1. Locating Device-specific Files



The **common** folder has system and header files specific to the device.

The **projects** folder has IDE-specific files, organized for each IDE. This may include linker and flash configuration files, settings files, and so on. It also has an empty `main.c` file. Within the **projects** folder, each IDE folder has a project file. Each project file is configured to use all the correct startup and configuration files, and the empty `main.c` file.

These project files do *not* use PDL-related files, nor do they have PDL-related include paths for header files.

What you do next depends upon your particular work flow. One reasonable approach is.

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

You can place the copy of the folder anywhere, because it is self-contained. For example, put it where you typically keep your development projects. Rename it if you wish.

2. **Copy `pdl_user.h` from a code example, and add it to your project.**

A good practice is to copy the file from a code example similar to your intended firmware. You must enable or disable the required driver functionality in your copy of `pdl_user.h`. See [Configuring the PDL](#).

3. **Include the PDL header file.**

In `main.c`, add a line of code to include `pdl_header.h`. This file includes `pdl_user.h`, so you do not need to explicitly include `pdl_user.h`.

```
#include "pdl_header.h"
```

#### 4. Add PDL source files to the project as required for your firmware.

Add the files you need from the PDL *drivers* or *utilities* folders. The code example projects have every source file. It is unlikely that you will need them all. Which files you require depends upon which features of the PDL you use.

#### 5. Update include paths to point to required header files.

The project files in the *lddevices* folder are not preconfigured with PDL source files or PDL-related include paths for a simple reason. You can locate your copy of this folder anywhere, and any preconfigured path would break. As a result, you need to update the include paths.

Precisely how to do this, and what paths you need to add or modify, will vary based on your circumstances. We assume you know how to add or edit paths. For each supported IDE, you can find the settings here:

- Atollic TrueSTUDIO: **Project > Build Settings > C/C++ General > Paths and Symbols**
- IAR Embedded Workbench: **Project > Options > C/C++ Compiler > Preprocessor**
- iSystem WinIDEA: **Project > Settings > Includes**
- Keil µVision: **Project > Options for Target > C/C++**
- GNU ARM Embedded Tools: edit the `directories` section of the makefile

Which paths you must modify or add depends upon which header files you need, and where they are located. Typically, you may need to add or modify these paths:

- Update or remove the path to *cmsis\include* (depends on whether you need these headers in your project)
- Add a path to *pdl\_user.h* (depends on where you put it)
- Add a path to the PDL *drivers* folder, < PDL directory>\drivers.
- Add a path to the PDL *utilities* folder, < PDL directory >\utilities

You do not need to modify the path to the *lcommon* folder, unless you restructure the project folder hierarchy.

If path-related errors continue to happen when you build your code (the actual error statement varies per IDE), add or modify the paths until your IDE can locate all the header files.

## 4.2 Configuring the PDL

To enable or disable a particular peripheral, modify *pdl\_user.h*. This file contains a long list of `#define` directives, one for each configurable feature of the PDL. To enable a peripheral, you define it as *PDL\_ON*. Otherwise it is *PDL\_OFF*. Code example projects already have these values set correctly for the specific example.

Figure 4-2. Enabling or Disabling a Peripheral

```

50  /**
51  ****
52  ** User Defines for PDL resource activation
53  **
54  ** Possible definitions are PDL_ON and PDL_OFF.
55  **
56  ****
57  // ADC
58  #define PDL_PERIPHERAL_ENABLE_ADC0      PDL_ON
59  #define PDL_PERIPHERAL_ENABLE_ADC1      PDL_OFF
60  #define PDL_PERIPHERAL_ENABLE_ADC2      PDL_OFF

```

If you do not enable a peripheral or feature that you use, the code will fail to compile because required header files are not included in the build process. Typical warnings refer to implicit function definitions. Typical errors refer to undefined terms. The precise warnings or errors depend upon your code and your IDE.

Similarly, if you enable a peripheral that is not supported on your target hardware, the PDL code will fail to compile for the same reason. In this case the PDL logic intentionally omits the required header files because the feature is not supported on the target hardware.

## 4.3 Configuring a Peripheral

The steps below outline the typical steps you perform to configure and initialize a peripheral. These steps occasionally vary. For example, some peripherals do not have a configuration structure.

### 1. Declare a configuration structure.

You need to know the C data type, and declare an instance of that data type.

```
stc_adc_config_t stcConfig; // declare an ADC configuration structure
```

### 2. Clear initial values.

The PDL provides a macro for the purpose, PDL\_ZERO\_STRUCT(x). For example,

```
PDL_ZERO_STRUCT(stcConfig); // zero all values
```

### 3. Fill in the fields.

Use values that configure the peripheral the way you want. For example, to use least significant bit alignment for the ADC results, write this code:

```
stcConfig.bLsbAlignment = TRUE;
```

### 4. Initialize the peripheral.

To actually set up the peripheral, call an *Init()* function. For example,

```
result = Adc_Init(&ADC0, &stcConfig); // initialize the ADC
```

You pass in the address of the peripheral instance, and the address of the configuration structure.

Many API function calls (such as this initialization call) require the base address of the registers used by the peripheral instance. PDL header files define constants for each available instance of a peripheral, such as ADC0 or ADC1. See



[Working with Registers](#) for more information about the base register address.

Note that initializing a peripheral does not start the peripheral. Most peripherals have an *enable* or *start* function in the API. Check the documentation for the peripheral you are using. For example, after calling *Adc\_Init()* you would use this code to enable the ADC, and wait until it is ready.

```
Adc_EnableWaitReady(&ADC0);
```

## 4.4 Using a Peripheral

After you configure and initialize a peripheral, you make API function calls to use the peripheral. In many cases the first call you make after initializing the peripheral will be to start or enable the peripheral. See the PDL documentation for guidance.

For example, to use the MFS peripheral, you may wish to accomplish the tasks in Table 4-1. If you look in the documentation, you can see what functions might accomplish the task, and learn how to call them.

Table 4-1. Example API Function Calls

MFS Tasks	PDL Function
Initialize	<i>Mfs_Uart_Init()</i>
Start	<i>Mfs_Uart_EnableFunc()</i>
Put a byte into the serial channel (transmit)	<i>Mfs_Uart_SendData()</i>
Read a byte from the serial channel (receive)	<i>Mfs_Uart_ReceiveData()</i>

## 4.5 Working with Registers

When programming using the PDL, you normally do not need to know about specific registers. The PDL handles that for you.

However, you do need to know the constant that represents the base address of the peripheral registers. In some cases the device supports several instances of a particular peripheral. For example, there can be multiple instances of MFS, which you may configure as a UART, an I<sup>2</sup>C, or a LIN. Each instance and mode has a distinct set of registers.

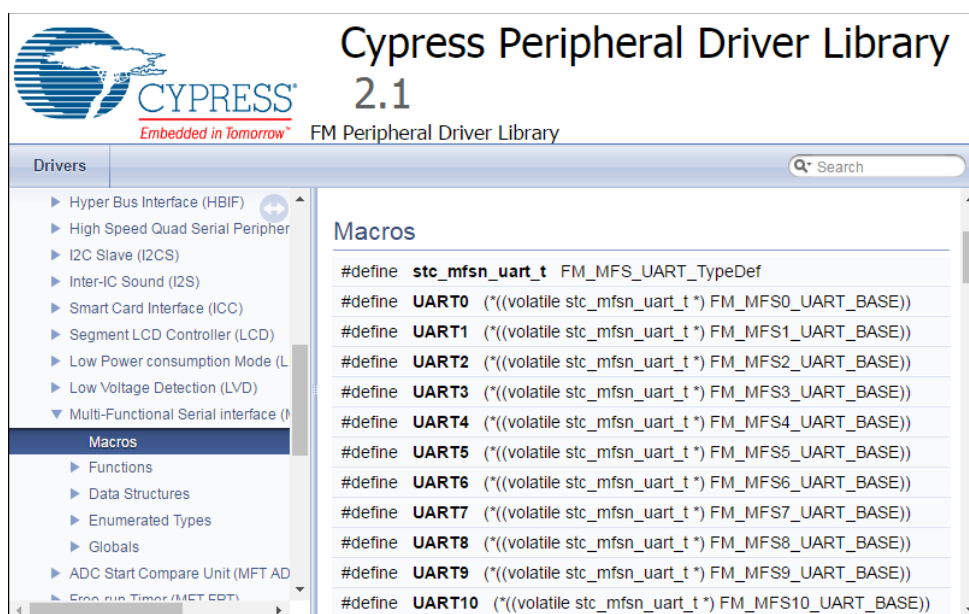
Many PDL API function calls require the base address of the registers for the instance. You pass this information by using a defined constant.

For example, this call initializes UART7. It passes the address of the instance, and the address of the configuration structure.

```
Mfs_Uart_Init(&UART7, &stcUartConfig);
```

Use the PDL documentation to get the correct defined constant.

Figure 4-3. Finding the Base Address for a Peripheral Instance



## 4.6 Managing Pins

Although the PDL handles register access for you via API function calls, your code may need to manage the state of individual pins. The file *gpio.h* defines macros for the purpose. If you use the macros you do need to know the pin, but do not need to know the pin registers.

You identify a pin by port and pin number. You can relate a particular physical pin (e.g. Pin 108) to a particular port (e.g. P1A) by examining the schematic for your kit, microcontroller, and package.

The macro name for pin management follows the form: `GPIO1PIN_P<port><pin>`.

For example, for Port 1, Pin A, the name is `GPIO1PIN_P1A`.

Table 4-2. Partial List of Pin-related Macros

Macro	Description
<code>Gpio1pin_InitIn(p,v)</code>	initialize pin as GPIO, input, pullup 0, set signal high or low (calling <code>Gpio1pin_InitVal()</code> to initialize the value)
<code>Gpio1pin_InitOut(p,v)</code>	initialize pin as GPIO, output, pullup 0, set signal high or low (calling <code>Gpio1pin_InitVal()</code> to initialize the value)
<code>Gpio1pin_InitVal(v)</code>	set signal to high or low; used as input to <code>Gpio1pin_InitIn()</code> , and <code>Gpio1pin_InitOut()</code> , so no pin specified
<code>Gpio1pin_Get(p)</code>	gets current signal (high or low) for the pin
<code>Gpio1pin_Put(p,v)</code>	set current signal (high or low) for the pin

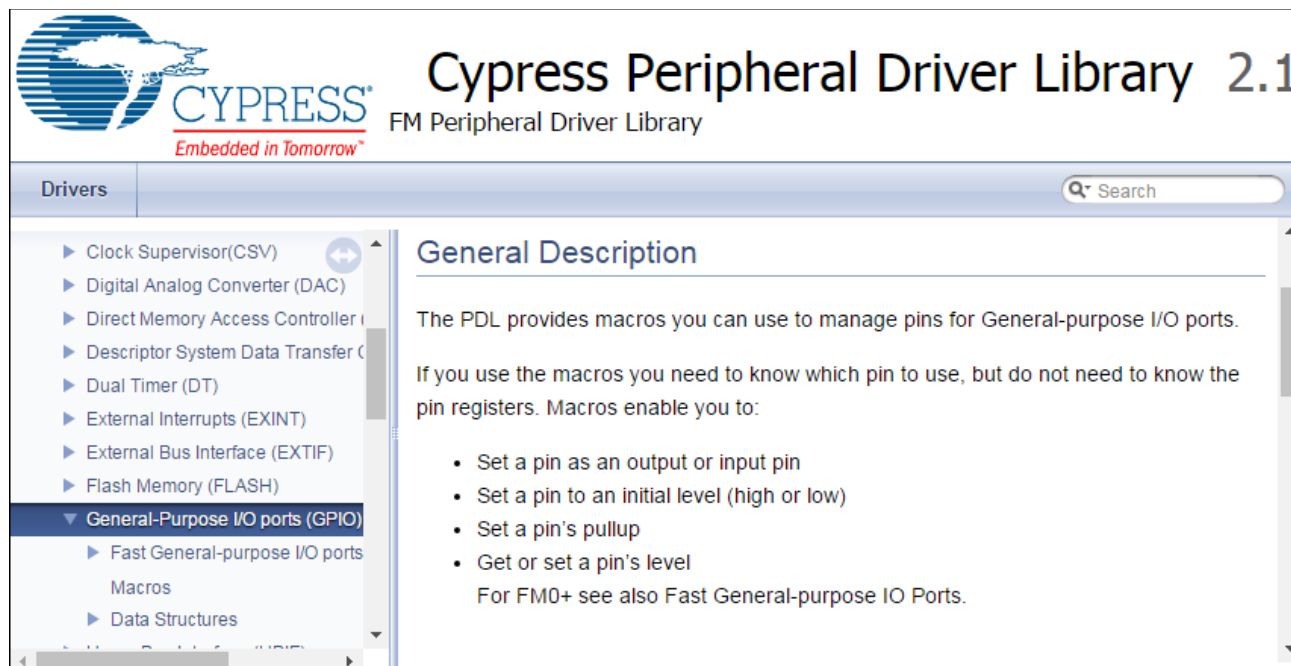
The parameter *p* is the name of the pin in question. The parameter *v* is the value, 0 for low, 1 for high. For example, you might write code like this:

```
Gpio1pin_InitOut(GPIO1PIN_P1A, Gpio1pin_InitVal( 1u ) );
```

This code initializes Port 1 Pin A to be an output pin, with signal high.

See the **General-Purpose I/O Ports** section of the PDL Documentation for more information.

Figure 4-4. GPIO Documentation



The screenshot shows the Cypress Peripheral Driver Library 2.1 documentation page. The left sidebar lists various drivers, with "General-Purpose I/O ports (GPIO)" selected. The main content area is titled "General Description" and contains the following text:

The PDL provides macros you can use to manage pins for General-purpose I/O ports.

If you use the macros you need to know which pin to use, but do not need to know the pin registers. Macros enable you to:

- Set a pin as an output or input pin
- Set a pin to an initial level (high or low)
- Set a pin's pullup
- Get or set a pin's level

For FM0+ see also Fast General-purpose IO Ports.

## 4.7 Using the PDL to Learn Low-level Programming

The PDL simplifies the embedded programming process by providing a high-level API. However, sometimes you may want to work at a low level. Working directly at a low level requires knowledge of registers and which bits control which features.

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

Suppose you want to enable (at a low level) a UART for transmitting data.

1. **Open a code example and locate an API function call that does what you want.**

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

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

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

3. **Examine the code.**

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

```
switch(enFunc)
{
    case UartTx:
        pstcUart->SCR_f.TXE = 1u;
        break;
```

Once you are familiar with naming conventions and coding style, this is easy to understand.

Table 4-3. Parsing PDL Source Code for Low-level Programming

Base address	Register	Bitfield	Value
pstcUart	SCR	TXE	bit is set to enable TX

You now know what register and bit to search for. You can use this information to locate details in the corresponding Peripheral Manual.

In your own low-level 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.

## 4.8 PDL Documentation

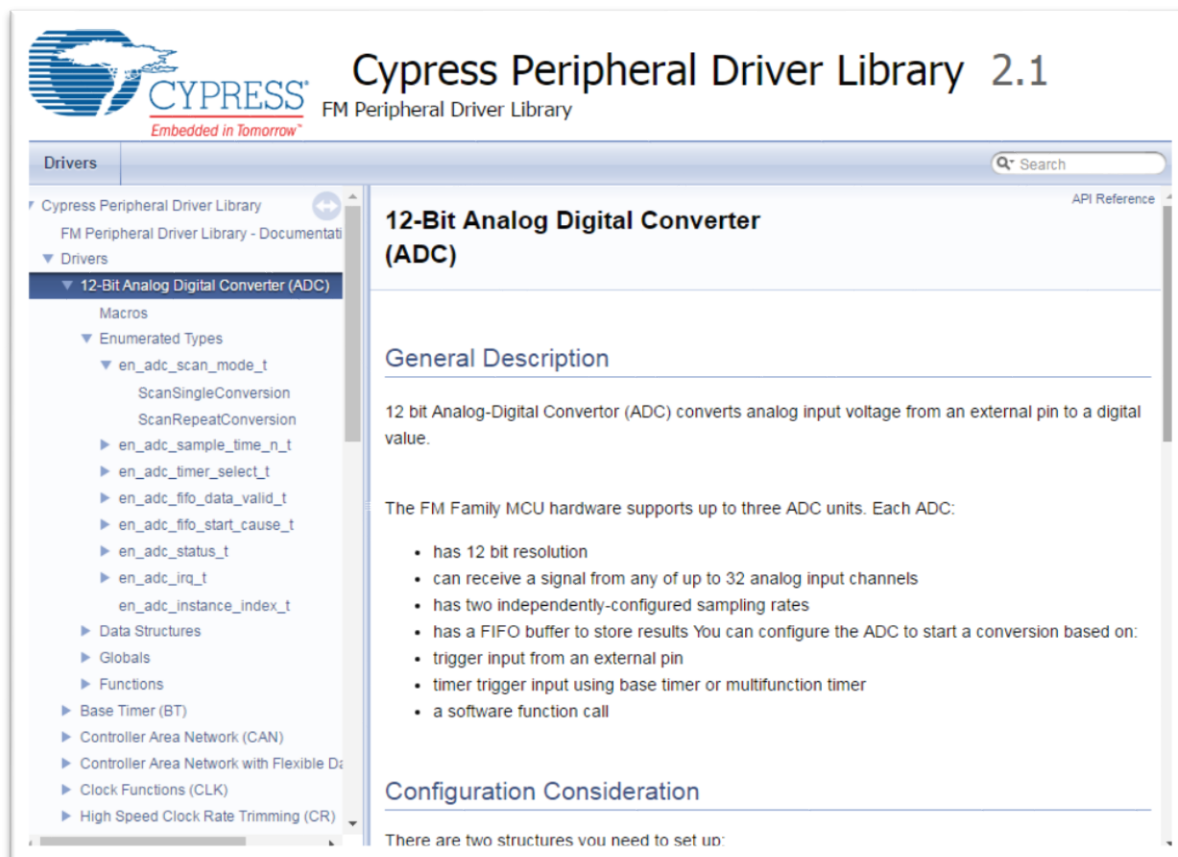
PDL documentation is HTML based and generated from source code. The PDL installer puts the documentation here:

<PDL directory>\doc\pdl\_api\_reference\_manual.html

The first time you open the documentation, make a bookmark in your browser for easy access.

In the documentation, use the left navigation menu to find the information you need. The **Drivers** section lists all the information for a particular peripheral. Expand any driver to see the macros, types, structures, global variables, and API functions.

Figure 4-5. PDL Documentation



## 4.9 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.

Table 4-4. PDL Naming Conventions

Naming Convention	Example	Description
Prefix stc_	stc_adc_config	This is a data structure.
Prefix en_	en_adc_status	This is an enumeration. When used as the field name for a data structure, this tells you that there are enumerations you should use to set the value for that field.
Leading characters in a variable/field name	bReturn u8Instance u16CompareValue u32Counter	The C data type for the variable: b = boolean u8 = uint8 and so forth.
Postfix _t	stc_adc_config_t	This is a data type, typically associated with a structure or enumeration.
<Peripheral>_<Function Name> ()	Adc_Enable()	An acronym for the peripheral, followed by the actual function name.

# A. PDL-Related Resources



Table A-4-5 lists some resources available to you for PDL Programming.

Table A-4-5. PDL and FM resources

I Want To	Resources
Evaluate the PDL	<p>Read this document.</p> <p>Purchase a Starter Kit that includes the PDL and PDL-based examples.</p> <ul style="list-style-type: none"> <li>■ <a href="#">FM0+ S6E1B8-Series Starter Kit</a>,</li> <li>■ <a href="#">FM0+ S6E1C3-Series Starter Kit</a>.</li> <li>■ <a href="#">FM4 S6E2GM Pioneer Kit</a></li> <li>■ <a href="#">FM4 S6E2H-Series Starter Kit</a></li> </ul> <p>Download the PDL from the <a href="#">Cypress PDL product page</a>.</p>
Learn About Available Tools	<p>IAR <a href="#">Embedded Workbench</a></p> <p>Keil <a href="#">µVision IDE</a></p> <p>iSYSTEM <a href="#">winIDEA</a></p> <p>Atollic <a href="#">TrueSTUDIO</a></p> <p><a href="#">GCC ARM Embedded</a></p>
Select an FM Part	<p>View the introductory video</p> <ul style="list-style-type: none"> <li>■ <a href="#">FM0+</a></li> <li>■ <a href="#">FM4</a></li> </ul> <p>Download and review the <a href="#">Product Selector Guide</a>.</p> <p>Read <a href="#">AN202487</a> - Differences Among FM0+, FM3, and FM4 Portfolios</p>
Become familiar with the PDL	<p>Read this document.</p> <p>Use the PDL documentation, located in your PDL installation.</p> <p>Explore the PDL code examples, located in your PDL installation.</p> <p>Explore the Starter Kit code examples, for kits that use the PDL.</p>
Become familiar with processor peripherals	<p>Use the available Peripheral Manuals as a technical reference.</p> <ul style="list-style-type: none"> <li>■ <a href="#">FM0+</a></li> <li>■ <a href="#">FM4</a></li> </ul> <p>Refer to the datasheet for your device.</p> <ul style="list-style-type: none"> <li>■ <a href="#">FM0+</a></li> <li>■ <a href="#">FM4</a></li> </ul>

# Revision History



## Document Revision History

Document Title: Cypress Peripheral Driver Library v2.1 Quick Start Guide			
Document Number: 002-13955			
Revision	Issue Date	Origin of Change	Description of Change
**	07/15/2016	JETT	Based on 002-04815, PDL 2.0 Quick Start Guide Rev *A, but this is a new document. Added information on changes between PDL 2.0 and 2.1 Updated Section 3 for new IDEs Added a troubleshooting section Added Section 4, Developing Code with the PDL Deleted Appendix B on Migrating a Keil project to V5 (not needed for v2.1)