CE220263 - PSoC 6 MCU GPIO Pins Example

Objective

This example demonstrates GPIO pin operation on the PSoC® 6 MCU, using ModusToolbox™ IDE.

Requirements

Tool: ModusToolbox™ IDE 1.1

Programming Language: C

Associated Parts: All PSoC 6 MCU parts

Related Hardware: PSoC 6 BLE Pioneer Kit, PSoC 6 WiFi-BT Pioneer Kit, PSoC 6 WiFi-BT Prototyping Kit

Overview

This example demonstrates multiple methods of configuring, reading, writing, and generating interrupts with PSoC 6 MCU General Purpose Input/Output (GPIO) pins. Both the ModusToolbox Device Configurator and PDL GPIO driver configuration methods are shown.

Hardware Setup

This example uses the kit's default configuration. Refer to the kit guide to ensure that the kit is configured correctly.

Note: The PSoC 6 BLE Pioneer kit and the PSoC 6 WiFi-BT Pioneer kit ship with KitProg2. ModusToolbox only works with KitProg3. Before using this code example, make sure that the kit is upgraded to KitProg3. See ModusToolbox **Help** > **ModusToolbox IDE Documentation** > **User Guide**; section "PSoC 6 MCU KitProg Firmware Loader". If you do not upgrade, you will see an error like "unable to find CMSIS-DAP device" or "KitProg firmware is out of date".

Software Setup

By default, this project uses ModusToolbox to configure the device hardware including pins. GPIO PDL driver pin and port configuration methods can be optionally enabled for some of the pins by setting the `PDL_PIN_CONFIGURATION` constant to (1u).

Operation

1. Connect the kit to your PC using the provided USB cable.
2. Import the code example into a new workspace. See KBA225201.
3. Build the application. Select the **mainapp** project in the project explorer. In the Quick Panel, click **Build Application**.
4. Program the PSoC 6 MCU device. In the project explorer, select the **mainapp** project. In the Quick Panel, scroll to the **Launches** section and click the **Program (KitProg3)** configuration.
5. Press KIT_BTN1 and observe the KIT_LED1 lights up while pressed, demonstrating GPIO read and write functions.
6. Release KIT_BTN1 and observe that KIT_LED1 turns OFF and KIT_LED2 turns ON for approximately 1 second demonstrating pin interrupt functionality.

Note: On the CY8CPROTO-062-WIFI-BT kit, only one LED is provided. KIT_LED1 is physically connected to the default projects KIT_LED2 pin. On Step 5 with this kit, no visible change will occur when KIT_BTN1 is pressed. On Step 6 with this kit, KIT_LED1 will respond as described for KIT_LED2. If you wish to demonstrate Step 5 operation and hide Step 6, rename KIT_LED1 to KIT_LED2, and KIT_LED2 to KIT_LED1 in the ModusToolbox Device Configurator Pins tab.

Debugging

You can debug the example to step through the code. Use the **Debug (KitProg3)** configuration. See KBA224621 to learn how to start a debug session with ModusToolbox IDE.

Design and Implementation

GPIO_Pins

This example demonstrates GPIO pin configuration, reading, writing, full port access, and interrupts, using GPIO PDL driver methods.

To demonstrate individual GPIO pin access, the example has a digital input pin connected to the development board user button, KIT_BTN1. The button state is continuously read by the CM4 processor. The value read is then written to a digital output pin connected to a kit LED. When the button is pressed the KIT_LED1 is ON. Several variations of reading and writing pin data are shown; select the method that works best for your design.

The digital input pin is also configured to generate an interrupt on a falling edge, which occurs on a button release. The interrupt routine causes the KIT_LED2 on a second digital output pin to turn ON for approximately 1 second.

A full port of GPIO pins is configured using the PDL Port Initialization function on Port 7. The value on Port 7 is continuously read with direct register reads, incremented, and written back to the port. The toggling port pins can be monitored on an oscilloscope.

Pin Configuration

Device configuration tools such as ModusToolbox Device Configurator automatically generate GPIO configuration code and execute it as part of the device boot process. GPIO PDL initialization methods are typically only used with manual PDL GPIO configuration when not using a configuration tool. They may also be used at run time to dynamically reconfigure GPIO pins independent of how the initial configuration was performed.

Most GPIO pins require only their basic parameters to be set and can use default values for all other settings. This allows the use of a simplified initialization function. `Cy_GPIO_Pin_FastInit()` only supports parameterized configuration of drive mode, output logic level, and High Speed Input/Output Multiplexer (HSIOM) setting. The HSIOM setting determines a pin's high-level software, peripheral, analog control, and connectivity. All other configuration settings are unchanged from their reset or previously set state. This function is very useful at run time to dynamically change a pin's configuration. For example, configure a pin to strong drive mode to write data, and then reconfigure the pin as High-Z to read data.

```
Cy_GPIO_Pin_FastInit(P1_5_PORT, P1_5_NUM, CY_GPIO_DM_STRONG, 1, HSIOM_SEL_GPIO);
```

A method to configure all attributes of a single pin is to use the `Cy_GPIO_Pin_Init()` function and a pin configuration structure. While easy to use, it generates larger code than other configuration methods.

```
Cy_GPIO_Pin_Init(P0_4_PORT, P0_4_NUM, &P0_4_Pin_Init);
```

The most code-efficient method to configure all attributes for a full port of pins is to use the `Cy_GPIO_Port_Init()` function and a port configuration structure. It packs all the configuration data into direct register writes for the whole port. Its limitation is that it must configure all pins in a port and the user must calculate the combined register values for all pins.

```
Cy_GPIO_Port_Init(GPIO_PRT7, &port7_Init);
```

Individual pin configuration settings can also be changed at run time using supplied driver functions. Examples of some of these functions are provided below. The function parameters demonstrate use of pin specific #defines provided in *cycfg_pins.h* when ModusToolbox is used.

```
Cy_GPIO_SetHSIOM(KIT_BTN1_PORT, KIT_BTN1_NUM, HSIOM_SEL_GPIO);
Cy_GPIO_SetDrivemode(KIT_BTN1_PORT, KIT_BTN1_NUM, CY_GPIO_DM_PULLUP);
Cy_GPIO_SetVtrip(KIT_BTN1_PORT, KIT_BTN1_NUM, CY_GPIO_VTRIP_CMOS);
Cy_GPIO_SetSlewRate(KIT_BTN1_PORT, KIT_BTN1_NUM, CY_GPIO_SLEW_FAST);
Cy_GPIO_SetDriveSel(KIT_BTN1_PORT, KIT_BTN1_NUM, CY_GPIO_DRIVE_FULL);
```

Pin Input Read Methods

The following methods all perform the same read from a GPIO pin using the different read methods available. Choose the most appropriate method for your specific use case. The `Cy_GPIO_Read()` function is thread- and multi-core-safe. Most GPIO driver functions require a minimum of two arguments to define the port and pin in that port. The port argument expects the base address of the port's registers. The pin argument expects the pin number within the port.

The preferred read method for use with ModusToolbox is using #defines provided by the configuration tool pin names. Pin name #defines are located in *\\<ProjectName>_config\GeneratedSource\cycfg_pins.h.*

```
pinReadValue = Cy_GPIO_Read(KIT_BTN1_PORT, KIT_BTN1_NUM);
```

The preferred read method for direct PDL use without a configuration tool is with user-defined custom #define pin names. They are typically placed in a user-created *.h* file.

```
#define mySwPin_Port P0_4_PORT
#define mySwPin_Num  P0_4_NUM
pinReadValue = Cy_GPIO_Read(mySwPin_Port, mySwPin_Num);
```

You can also read pins using default device pin name #defines provided for each device and package. The default name #defines are located in *\\<ProjectName>_mainapp_(device series)pdl\Includes\(..devices/ series/include)\gpio_(series+package).h*.

```
pinReadValue = Cy_GPIO_Read(P0_4_PORT, P0_4_NUM);
```

Pins can also be read using default port register name #defines and pin number #defines located in *\\<Project Name>_mainapp_(device series)pdl\Includes\(..devices/series/include)\(part number).h.*

```
pinReadValue = Cy_GPIO_Read(GPIO_PRT0, 4);
```

Pin reads using port and pin numbers are also supported. This method is useful for algorithmically generated port and pin numbers. `Cy_GPIO_PortToAddr()` is a helper function that converts the port number into the required port register base address required by other GPIO driver functions.

```
portNumber = 0;
pinReadValue = Cy_GPIO_Read(Cy_GPIO_PortToAddr(portNumber), 4);
```

Like any MCU, direct port register access is always available and useful for accessing multiple pins in a port simultaneously or developing application-optimized port accesses. The following example shows a port IN register read with mask and shift of the desired pin data.

```
pinReadValue = (GPIO_PRT0->IN >> P0_4_NUM) & CY_GPIO_IN_MASK;
```

Pin Output Write Methods

The following methods all perform the same write to GPIO pins using the different write methods available. Choose the most appropriate method for your specific use case. The `Cy_GPIO_Write()` function is best used when the desired pin state is not already known and is determined at run time. The write function uses atomic operations that directly affect only the selected pin without using read-modify-write operations. The write function is therefore thread- and multi-core-safe.

The preferred write method for use with ModusToolbox is using #defines provided by the configuration tool pin names. The generated #defines are located in *\\<ProjectName>_config\GeneratedSource\cycfg_pins.h*.

```
Cy_GPIO_Write(KIT_BTN1_PORT, KIT_BTN1_NUM, pinReadValue);
```

The preferred method for direct PDL use without a configuration tool is pin writes with user-defined custom #define pin names. The user supplied #defines are typically placed in a user-generated *.h* file.

```
#define myLedPin_Port P1_5_PORT
#define myLedPin_Num  P1_5_NUM
Cy_GPIO_Write(myLedPin_Port, myLedPin_Num, pinReadValue);
```

Pin writes can also be made using the default device pin name #defines located in the *\\<ProjectName>_mainapp_(device series)pdl\Includes\(..devices/series/include)\gpio_(series+package).h file.*

```
Cy_GPIO_Write(P1_5_PORT, P1_5_NUM, pinReadValue);
```

Pin writes are possible using default port register name #defines and pin numbers. #defines located in the *\\<ProjectName>_mainapp_(device series)pdl\Includes\(..devices/series/include)\(part number) .h file.*

```
Cy_GPIO_Write(GPIO_PRT1, 5, pinReadValue);
```

For algorithmically generated port and pin numbers, pin writes using port and pin numbers are very useful. `Cy_GPIO_PortToAddr()` is a helper function that converts the port number into the required port register base address:

```
portNumber = 1;
Cy_GPIO_Write(Cy_GPIO_PortToAddr(portNumber), 5, pinReadValue);
```

The most efficient output methods, when the desired pin state is already known at compile time, are to directly Set, Clear, and Invert the pin output state. These register writes are atomic operations that directly affect only the selected pin without using read-modify-write operations. They are therefore thread- and multi-core-safe. The same argument variations as demonstrated with the `Cy_GPIO_Write()` function can be used.

```
Cy_GPIO_Set(KIT_LED1_PORT, KIT_LED1_NUM);
Cy_GPIO_Clr(KIT_LED1_PORT, KIT_LED1_NUM);
Cy_GPIO_Inv(KIT_LED1_PORT, KIT_LED1_NUM);
```

Port Access

Direct register access is used to interface with multiple pins in one port at the same time. These accesses may not be thread- or multi-core-safe due to possible read-modify-write operations. All pins in a port under direct register control should only be accessed by a single CPU core unless access protections are provided at the system level.

```
portReadValue = GPIO_PRT7->IN;
portReadValue++;
GPIO_PRT7->OUT = portReadValue;
```

Pin Interrupts

To generate a pin interrupt, configure it to trigger on a rising, falling, or both edges, and mask it so that the pin signal is sent to the interrupt controller vector for that port.

```
Cy_GPIO_SetInterruptEdge(KIT_BTN1_PORT, KIT_BTN1_NUM, CY_GPIO_INTR_RISING);
Cy_GPIO_SetInterruptMask(KIT_BTN1_PORT, KIT_BTN1_NUM, CY_GPIO_INTR_EN_MASK);
```

The port interrupt vector must then be configured, cleared, and enabled to be triggered from the port interrupt signal and mapped to the desired Interrupt Service Routine (ISR). See the PDL Cy_SysInt documentation for more information on interrupt configuration and use.

```
Cy_SysInt_Init(&intrCfg, GPIO_Interrupt);
NVIC_ClearPendingIRQ(intrCfg.intrSrc);
NVIC_EnableIRQ((IRQn_Type)intrCfg.intrSrc);
```

After an interrupt occurs, the pin interrupt must be cleared before exiting the ISR so that the edge detection logic is reset to allow the detection of the next edge.

```
void GPIO_Interrupt()
{

    Cy_GPIO_ClearInterrupt(KIT_BTN1_PORT, KIT_BTN1_NUM);
}
```

If more than one pin in a port can generate an interrupt, the `Cy_GPIO_GetInterruptStatus()` function may be used to identify the pin that detected an edge event and generated the interrupt. Optionally, direct register reads of the INTR register may be used to determine the interrupt pin.

```
Cy_GPIO_ClearInterrupt(KIT_BTN1_PORT, KIT_BTN1_NUM);

portIntrStatus = KIT_BTN1_PORT->INTR;
if(CY_GPIO_INTR_STATUS_MASK == ((portIntrStatus >> KIT_BTN1_NUM) &
                                CY_GPIO_INTR_STATUS_MASK))
```

In most designs, each port that can generate an interrupt will be assigned to its own interrupt vector. If interrupt vector limitations do not allow discrete vectors to be used, the combined port interrupt (AllPortInt) may be used. The AllPortInt ORs all of the individual port interrupt signals into a single chip-wide signal requiring only one vector for any number of pins across multiple ports. The `Cy_GPIO_GetInterruptCause0()` function can be used to identify the port(s) that detected interrupt edge events.

```
if((Cy_GPIO_GetInterruptCause0() & INTCAUSE0_PORT0) != 0)
```

Resources and Settings

Table 1 lists some of the ModusToolbox resources used in the example, and how they are used in the design. The *design.modus* file contains all of the configuration settings. For example, for pin usage and configuration, open the **Pins** tab of the design file.

Table 1. ModusToolbox Resources

Resource	Alias	Purpose	Non-default Settings
Pin	KIT_BTN1	Input pin to read a switch value. When the button is pressed, the switch shorts the pin to ground. The resistive pull-up drive mode causes a logic 1 to be read when the button is released.	Drive Mode = Resistive Pull-up. Input buffer on Interrupt Trigger Type = Falling Edge
Pin	KIT_LED1	Output pin to drive an LED	Drive Mode = Strong Drive. Input buffer off
Pin	KIT_LED2	Output pin to drive an LED, to show interrupt trigger	Drive Mode = Strong Drive. Input buffer off

Reusing This Example

This example is configured for the supported kit(s). To port the design to a different PSoC 6 MCU device, right-click an application project and choose **Change Device**. If changing to a different kit, you may need to reassign pins. If pins are reassigned, the code examples that use physical port and pin numbers will require updating.

Table 2. Device and Pin Mapping across PSoC 6 MCU Kits

Kit Name	Device Used	KIT_BTN1	KIT_LED1	KIT_LED2
CY8CKIT-062-WiFi-BT	CY8C6247BZI-D54	SW2 P0[4]	LED8 P1[5]	LED9 P13[7]
CY8CKIT-062-BLE	CY8C6347BZI-BLD53	SW2 P0[4]	LED8 P1[5]	LED9 P13[7]
CY8CPROTO-062-4343W	CY8C624ABZI-D44	SW2 P0[4]	-	LED4 P13[7]

Note: On the CY8CPROTO-062-WIFI-BT kit, only one LED is provided. KIT_LED1 is physically connected to the default projects KIT_LED2 pin. If you wish to demonstrate default KIT_LED1 operation, rename KIT_LED1 to KIT_LED2, and KIT_LED2 to KIT_LED1 in the ModusToolbox Device Configurator Pins tab.

Related Documents

Application Notes	
AN221774 – Getting Started with PSoC 6 MCU	Describes PSoC 6 MCU devices and how to build your first ModusToolbox and PSoC Creator projects
AN210781 – Getting Started with PSoC 6 MCU with Bluetooth Low Energy (BLE) Connectivity	Describes PSoC 6 MCU with BLE Connectivity devices and how to build your first PSoC Creator project
AN215656 – PSoC 6 MCU: Dual-CPU System Design	Describes the dual-CPU architecture in PSoC 6 MCU, and shows how to build a simple dual-CPU design
Code Examples	
Visit the Cypress GitHub site for a comprehensive collection of code examples using ModusToolbox IDE	
Device Documentation	
PSoC 6 MCU Datasheets	PSoC 6 MCU Technical Reference Manuals
Development Kits	
CY8CKIT-062-BLE PSoC 6 BLE Pioneer Kit	
CY8CKIT-062-WiFi-BT PSoC 6 WiFi-BT Pioneer Kit	
CY8CPROTO-062-4343W PSoC 6 Wi-Fi BT Prototyping Kit	

Tool Documentation	
ModusToolbox IDE	The Cypress IDE for IoT designers

Cypress Resources

Cypress provides a wealth of data at www.cypress.com to help you to select the right device, and quickly and effectively integrate the device into your design.

For PSoC 6 MCU devices, see KBA223067 in the Cypress community for a comprehensive list of PSoC 6 MCU resources.

Document History

Document Title: CE220263 - PSoC 6 MCU GPIO Pins Example

Document Number: 002-25441

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	6365953	GJV	11/22/2018	New code example
*A	6490828	GJV	02/22/2018	Update for ModusToolbox 1.1

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Code Examples | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709