APPLICATION NOTE

PAC52XX GPIO Peripheral Firmware Design

Power Application Controller™

Marc Sousa
Senior Manager, Systems and Firmware

www.active-semi.com

TABLE OF CONTENTS

OVERVIEW

The PAC52XX family of controllers have a power set of peripherals to ease design of mixed-signal control and other embedded systems. Included in this family are a highly configurable set of GPIO that can be used by the application.

The IO pins on the PAC52XX can be used for both the other peripherals (timers, ADC, serial communications) or for program defined input/output.[1]

The number of IO pins on each device in the PAC52XX family may vary. There are up to 5 GPIO ports, each with up to 8 pins that may be used by the application. The GPIO ports are named as follows:

- GPIOA
- GPIOB
- GPIOC
- GPIOD
- GPIOE

GPIOC are used for analog input for the ADC, and have special configuration and limited features for GPIO. For more information on configuring GPIOC for analog input, see the PAC52XX User Guide, or application notes on the PAC52XX ADC.

When used as GPIO, the IO pins have the following features:

- May be configured as digital input or output
- Configurable drive strength (except GPIOC)
- Configurable weak pull-up or pull-down to VDDIO (except GPIOC)
- Configurable interrupts

The chapters that follow will describe how to implement firmware to take advantage of the GPIO features.

In the firmware examples in this application note, two types of firmware are shown for each example: PAC52XX peripheral registers and the PAC52XX SDK.

[1] For information on how other peripherals use the GPIO when in peripheral mode, see the PAC52XX User Guide, or other application notes on this topic.

PAC52XX GPIO PERIPHERAL REGISTERS

The PAC52XX Peripheral Registers are a representation of all the PAC52XX registers as a set of data structures. Complete registers can be written to, as well as portions of registers (*e.g.*, selected bits) using C-language bit-fields.

The GPIO peripheral register header file is named `pac5xxx_gpio.h` and is located in the SDK installation directory. Below are the contents of this peripheral header file for reference:

```
typedef union
{
  __IO uint32_t w;
  struct {
  __IO uint32_t s                   : 16;
       uint32_t                     : 16;
  };
  struct {
  __IO uint32_t b                   : 8;
       uint32_t                     : 24;
  };
  struct {
    __IO uint32_t P0                : 1;          /*!< P0 (bit 0)                                          */
    __IO uint32_t P1                : 1;          /*!< P1 (bit 1)                                          */
             __IO uint32_t P2               : 1;          /*!< P2 (bit 2)                                          */
             __IO uint32_t P3               : 1;          /*!< P3 (bit 3)                                          */
             __IO uint32_t P4               : 1;          /*!< P4 (bit 4)                                          */
             __IO uint32_t P5               : 1;          /*!< P5 (bit 5)                                          */
             __IO uint32_t P6               : 1;          /*!< P6 (bit 6)                                          */
             __IO uint32_t P7               : 1;          /*!< P7 (bit 7)                                          */
                  uint32_t                  : 24;
  };
  struct {
             __IO uint32_t LO_P0            : 1;          /*!< PY (bit 0) for low port                             */
             __IO uint32_t LO_P1            : 1;          /*!< P1 (bit 1) for low port                             */
             __IO uint32_t LO_P2            : 1;          /*!< P2 (bit 2) for low port                             */
             __IO uint32_t LO_P3            : 1;          /*!< P3 (bit 3) for low port                             */
             __IO uint32_t LO_P4            : 1;          /*!< P4 (bit 4) for low port                             */
             __IO uint32_t LO_P5            : 1;          /*!< P5 (bit 5) for low port                             */
             __IO uint32_t LO_P6            : 1;          /*!< P6 (bit 6) for low port                             */
             __IO uint32_t LO_P7            : 1;          /*!< P7 (bit 7) for low port [e.g., for GPIOXY, this is GPIOX7  */
                  uint32_t                  : 8;
             __IO uint32_t HI_P0            : 1;          /*!< P6 (bit 0) for high port                            */
             __IO uint32_t HI_P1            : 1;          /*!< P6 (bit 1) for high port                            */
             __IO uint32_t HI_P2            : 1;          /*!< P6 (bit 2) for high port                            */
             __IO uint32_t HI_P3            : 1;          /*!< P6 (bit 3) for high port                            */
             __IO uint32_t HI_P4            : 1;          /*!< P6 (bit 4) for high port                            */
             __IO uint32_t HI_P5            : 1;          /*!< P6 (bit 5) for high port                            */
             __IO uint32_t HI_P6            : 1;          /*!< P6 (bit 6) for high port                            */
             __IO uint32_t HI_P7            : 1;          /*!< P7 (bit 7) for high port [e.g., for GPIOXY, this is GPIOY7 */
          uint32_t                  : 8;
  };
} PAC5XXX_GPIO_PINS_TypeDef;

#define GPIO_DS_8MA             0           /*!< GPIO Drive Strength: 8mA                                */
#define GPIO_DS_16MA            1           /*!< GPIO Drive Strength: 16mA                               */

typedef union
{
  __IO uint32_t w;
  struct {
  __IO uint32_t s                   : 16;
       uint32_t                     : 16;
  };
  struct {
  __IO uint32_t b                   : 8;
       uint32_t                     : 24;
  };
  struct {
    __IO uint32_t P0                : 2;          /*!< P0 (bit 0)                                          */
    __IO uint32_t P1                : 2;          /*!< P1 (bit 1)                                          */
             __IO uint32_t P2               : 2;          /*!< P2 (bit 2)                                          */
             __IO uint32_t P3               : 2;          /*!< P3 (bit 3)                                          */
             __IO uint32_t P4               : 2;          /*!< P4 (bit 4)                                          */
             __IO uint32_t P5               : 2;          /*!< P5 (bit 5)                                          */
             __IO uint32_t P6               : 2;          /*!< P6 (bit 6)                                          */
             __IO uint32_t P7               : 2;          /*!< P7 (bit 7)                                          */
                  uint32_t                  : 16;
  };
  struct {
             __IO uint32_t LO_P0            : 2;          /*!< PY (bit 0) for low port                             */
             __IO uint32_t LO_P1            : 2;          /*!< PY (bit 1) for low port                             */
             __IO uint32_t LO_P2            : 2;          /*!< PY (bit 2) for low port                             */
             __IO uint32_t LO_P3            : 2;          /*!< PY (bit 3) for low port                             */
             __IO uint32_t LO_P4            : 2;          /*!< PY (bit 4) for low port                             */
             __IO uint32_t LO_P5            : 2;          /*!< PY (bit 5) for low port                             */
             __IO uint32_t LO_P6            : 2;          /*!< PY (bit 6) for low port                             */
             __IO uint32_t LO_P7            : 2;          /*!< PY (bit 7) for low port [e.g., for GPIOYZ, this is GPIOY7  */
             __IO uint32_t HI_P0            : 2;          /*!< PZ (bit 0) for high port                            */
```

```
    __IO uint32_t HI_P1          : 2;          /*!< PZ (bit 1) for high port                                  */
    __IO uint32_t HI_P2          : 2;          /*!< PZ (bit 2) for high port                                  */
    __IO uint32_t HI_P3          : 2;          /*!< PZ (bit 3) for high port                                  */
    __IO uint32_t HI_P4          : 2;          /*!< PZ (bit 4) for high port                                  */
    __IO uint32_t HI_P5          : 2;          /*!< PZ (bit 5) for high port                                  */
    __IO uint32_t HI_P6          : 2;          /*!< PZ (bit 6) for high port                                  */
    __IO uint32_t HI_P7          : 2;          /*!< PZ (bit 7) for high port [e.g., for GPIOYZ, this is GPIOZ7 */
  };
} PAC5XXX_GPIO_PSEL_TypeDef;


typedef struct
{
  __IO PAC5XXX_GPIO_PINS_TypeDef OUT;          /*!< Offset: 0x0000   GPIO Output Register                     */
  __IO PAC5XXX_GPIO_PINS_TypeDef OUTEN;        /*!< Offset: 0x0004   GPIO Output Enable Register              */
  __IO PAC5XXX_GPIO_PINS_TypeDef DS;           /*!< Offset: 0x0008   GPIO Output Drive Strength Register      */
  __IO PAC5XXX_GPIO_PINS_TypeDef PU;           /*!< Offset: 0x000C   GPIO Output Weak Pull-up Register        */
  __IO PAC5XXX_GPIO_PINS_TypeDef PD;           /*!< Offset: 0x0010   GPIO Output Weak Pull-down Register      */
  __I  PAC5XXX_GPIO_PINS_TypeDef IN;           /*!< Offset: 0x0014   GPIO Input Register                      */
  __IO PAC5XXX_GPIO_PINS_TypeDef INE;          /*!< Offset: 0x0024   GPIO Input Enable Register               */
  __IO PAC5XXX_GPIO_PSEL_TypeDef PSEL;         /*!< Offset: 0x001C   GPIO Peripheral Select Register          */
  __IO PAC5XXX_GPIO_PINS_TypeDef INTP;         /*!< Offset: 0x0020   GPIO Interrupt Polarity Select Register  */
  __IO PAC5XXX_GPIO_PINS_TypeDef INTE;         /*!< Offset: 0x0024   GPIO Interrupt Enable Register           */
  __IO PAC5XXX_GPIO_PINS_TypeDef INTF;         /*!< Offset: 0x0028   GPIO Interrupt Flag Register             */
  __IO PAC5XXX_GPIO_PINS_TypeDef INTM;         /*!< Offset: 0x002C   GPIO Interrupt Mask Register             */
} PAC5XXX_GPIO_TypeDef;
```

As seen above, the peripheral header file defines all of the registers for each GPIO port so they can be accessed efficiently by the application firmware.

The structure of all 5 GPIO ports is the same, and each GPIO port can be referenced in firmware using pointers as shown below. Each GPIO port has a type of *PAC5XXX_GPIO_TypeDef.*

GPIOA	PAC5XXX_GPIOA->*[reg.field_name]*
GPIOB	PAC5XXX_GPIOB->*[reg.field_name]*
GPIOC	PAC5XXX_GPIOC->*[reg.field_name]*
GPIOD	PAC5XXX_GPIOD->*[reg.field_name]*
GPIOE	PAC5XXX_GPIOE->*[reg.field_name]*

For example, the following fields in the following registers may be addressed as shown below.

GPIOA pin 5 output enable: `PAC5XXX_GPIOA->OUTEN.P5`

GPIOD pin 7 input state: `PAC5XXX_GPIOD->IN.P7`

GPIOB (all pins) output state: `PAC5XXX_GPIOB->OUT.b`

The *field_name* represents how the register will be read or written by the program. The *field_name* may either be a bit-field (one or more bits that be referenced by a single name) or the entire register. The following conventions are used in the PAC52XX Peripheral Header files:

Field Name	Access
w	32-bit word (reads register as a 32-bit word)
s	16-bit word (reads register as a 16-bit word)
b	8-bit unsigned character
P0	Single bit (pin 0)
P1	Single bit (pin 1)
P2	Single bit (pin 2)
P3	Single bit (pin 3)
P4	Single bit (pin 4)
P5	Single bit (pin 5)
P6	Single bit (pin 6)
P7	Single bit (pin 7)

PAC52XX GPIO SDK

The PAC52XX SDK is a C-Language interface that allows higher-level programming of the PAC52XX, without having to know the details of the register memory map, and bit positions within registers.

The SDK is most useful for configuration, where there are many registers that need to be written.

The definition of the PAC52XX GPIO SDK is located in the file `pac5xxx_driver_gpio.h`. In this file, the user can call C-language functions to configure and operate the GPIO.

When possible, the SDK uses inline functions to implement the functions for simple register reads and writes, to maintain performance. Other functions that perform many register reads and writes contain the function inside a *.c file, that is a C-callable function.

For example, to check to see if the interrupt flag is set for any pin in GPIOA the user may call this function:

```
RAMFUNC_GPIO static  uint8_t pac5xxx_gpio_int_flag_set_a(void) { return PAC5XXX_GPIOA->INTF.b; }
```

Since this is a single register read, then this is just as efficient as using the peripheral header files, but the user does not need to know details of register names.

For more complex functions, the user would call a C-language function. For example, the function to configure interrupts on GPIOB is shown below:

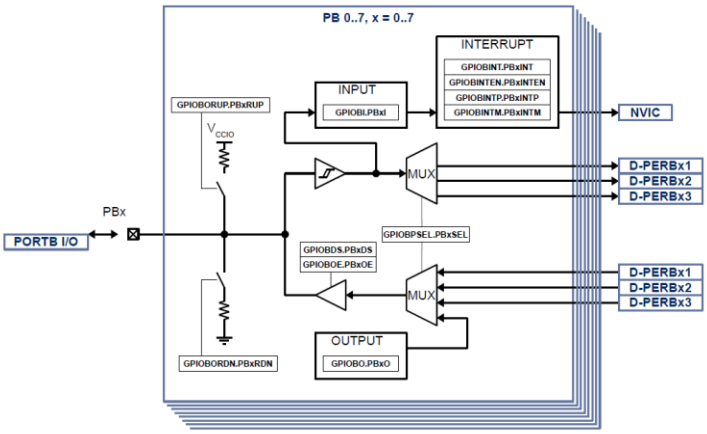
```
RAMFUNC_GPIO void pac5xxx_gpio_configure_interrupt_b(uint8_t port_mask, uint8_t active_high_mask);
```

In the examples below, both types of techniques are described.

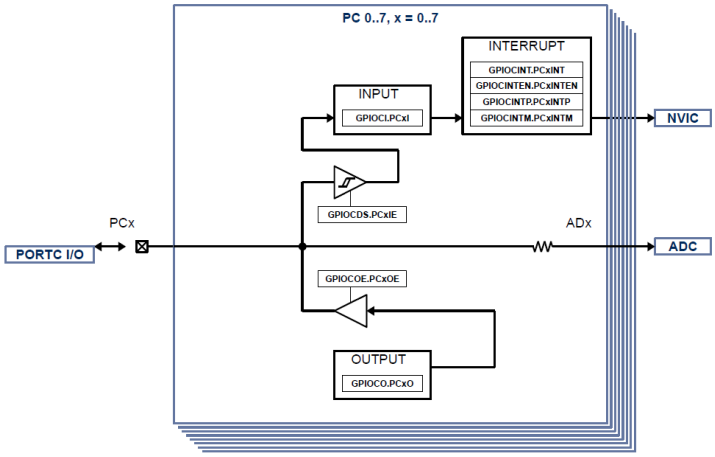
GPIO Block Diagram

Each PAC52XX device has a different number of GPIO ports and pins available. However, the same design for each GPIO (exception GPIOC) is the same.

The block diagram for each GPIO port is similar to the one shown below, for GPIOB. This will be the same for GPIOA, GPIOB, GPIOD and GPIOE.



GPIOC has a different design, since it is also used for analog input. The block diagram for GPIOC is shown below:



As shown above, GPIOC does not have support for the drive strength or weak pull-up/pull-down as the other GPIO ports.

DIGITAL OUTPUT CONFIGURATION

To use a GPIO pin for digital output (under firmware control), the output enable signal must be set to on and then the output state may be used. As soon as the output enable signal is set to on, the pin will be driven to the state specified by the output state register (0 by default) so care should be taken to configure the output state before turning on the output enable.

Note that for GPIOC, the input enable signal may be used for analog input, and must be cleared for this operation. To perform the following actions, the code below may be used.

Set GPIOA, pin 5 and to logic high:

[Peripheral]

```
PAC5XXX_GPIOA->OUT.P5 = 1;
PAC5XXX_GPIOA->OUTEN.P5 = 1;
```

[SDK]

```
pac5xxx_gpio_out_pin_set_a(5);
pac5xxx_gpio_out_enable_a(0x20); // sets all pins with this mask
```

Set GPIOD, all pins to logic low:

[Peripheral]

```
PAC5XXX_GPIOD->OUT.b = 0;
PAC5XXX_GPIOD->OUTEN.b = 0xFF;
```

[SDK]

```
pac5xxx_gpio_out_d(0);
pac5xxx_gpio_out_enable_a(0xFF);
```

Set GPIOC, P2 and P3 to logic high:

[Peripheral]

```
PAC5XXX_GPIOC->OUT.P2 = 1;
PAC5XXX_GPIOC->OUT.P3 = 1;
PAC5XXX_GPIOC->INE.P2 = 0;
PAC5XXX_GPIOC->INE.P3 = 0;
PAC5XXX_GPIOC->OUTEN.P2 = 1;
PAC5XXX_GPIOC->OUTEN.P3 = 1;
```

[SDK]

```
pac5xxx_gpio_out_pin_set_a(2);
pac5xxx_gpio_out_pin_set_a(3);
pac5xxx_gpio_digital_enable_c(0x0C); // sets all pins with this mask
pac5xxx_gpio_out_enable_a(0xC0);  // sets all pins with this mask
```

DIGITAL INPUT CONFIGURATION

To use a GPIO pin for digital input (under firmware control), the output enable signal must be set to off (0) and then the input state may be used. The input state is always available, if the pin is set to be an input or an output, so it may always be sampled.

Note that for GPIOC, the input enable signal is used for analog input, and must be set for this operation. To perform the following actions, the code below may be used.

Sample GPIOA, pin 5:

[Peripheral]

```
PAC5XXX_GPIOA->OUTEN.P5 = 0;
state = PAC5XXX_GPIOA->IN.P5;
```

[SDK]

```
pac5xxx_gpio_out_enable_a(0x20); // sets all pins with this mask
state = pac5xxx_gpio_in_pin_a(5);
```

Sample GPIOC, pin 2:

[Peripheral]

```
PAC5XXX_GPIOC->OUTEN.P2 = 0;
PAC5XXX_GPIOC->INE.P2 = 1;
state = PAC5XXX_GPIOC->IN.P2;
```

[SDK]

```
pac5xxx_gpio_out_enable_a(0x00); // clears all pins with this mask
pac5xxx_gpio_digital_enable_c(0x04); // sets all pins with this mask
state = pac5xxx_gpio_in_pin_c(2);
```

ANALOG INPUT CONFIGURATION

To use a GPIO pin for analog input only GPIOC may be used. To configure this behavior, the output enable signal must be set to off (0) and the input enable signal must also be set to off (0).

To perform the following actions, the code below may be used.

Use GPIOC, pin 0-3 for ADC:

[Peripheral]

```
PAC5XXX_GPIOC->OUTEN.P0 = 0;
PAC5XXX_GPIOC->OUTEN.P1 = 0;
PAC5XXX_GPIOC->OUTEN.P2 = 0;
PAC5XXX_GPIOC->OUTEN.P3 = 0;
PAC5XXX_GPIOC->INEN.P0 = 0;
PAC5XXX_GPIOC->INEN.P1 = 0;
PAC5XXX_GPIOC->INEN.P2 = 0;
PAC5XXX_GPIOC->INEN.P3 = 0;
```

[SDK]

```
pac5xxx_gpio_out_enable_a(0); // sets all pins with this mask
pac5xxx_gpio_digital_enable_c(0); // sets all pins with this mask
```


DRIVE STRENGTH CONFIGURATION

In the PAC52XX, any available GPIOA, GPIOB, GPIOD and GPIOE pin drive strength may be configured between low-current drive strength and high-current drive strength. GPIOC does not have this feature. For the specific drive strength for each of these settings, see the PAC52XX data sheet for then device used.

Of course, the user must also set the output enable signal for the given ports and pins to use this feature.

To perform the following actions, the code below may be used.

Set GPIOA, pin 5 to high-current:

[Peripheral]

```
PAC5XXX_GPIOA->DS.P5 = 1;
PAC5XXX_GPIOA->OUTEN.P5 = 0;
```

[SDK]

```
pac5xxx_gpio_out_drive_strength_a(0x20); // sets all pins with this mask
pac5xxx_gpio_out_enable_a(0x20); // sets all pins with this mask
```

Set GPIOB, all pins to low-current:

[Peripheral]

```
PAC5XXX_GPIOB->DS.b = 0;
PAC5XXX_GPIOA->OUTEN.b = 0xFF;
```

[SDK]

```
pac5xxx_gpio_out_drive_strength_b(0); // sets all pins with this mask
pac5xxx_gpio_out_enable_b(0xFF); // sets all pins with this mask
```

WEAK PULL-UP AND PULL-DOWN CONFIGURATION

In the PAC52XX, any available GPIOA, GPIOB, GPIOD and GPIOE pin may be configured to have a weak pull-up, weak pull-down or neither. GPIOC does not have this feature.

Of course, this feature should be used when configuring the pin as a digital input. Also, the user needs to make sure that the pull-up and pull-down bits are not both enabled at the same time.

To perform the following actions, the code below may be used.

Set GPIOA, pin 5 weak pull-up:

[Peripheral]

```
PAC5XXX_GPIOA->PD.P5 = 0;         // Set pull-down to disabled
PAC5XXX_GPIOA->PU.P5 = 1;         // Set pull-up to enabled
PAC5XXX_GPIOA->OUTEN.P5 = 0;      // Set pin to digital input
state = PAC5XXX_GPIOA->IN.P5;     // Sample input state
```

[SDK]

```
pac5xxx_gpio_out_pull_down_a(0);    // sets all pins in port with this mask
pac5xxx_gpio_out_pull_up_a(0x20);   // sets all pins in port with this mask
pac5xxx_gpio_out_enable_a(0);       // sets all pins in port with this mask
```

Set GPIOB, all pins weak pull-down:

[Peripheral]

```
PAC5XXX_GPIOB->PU.b = 0;          // Set all pins pull-up to disabled
PAC5XXX_GPIOB->PD.b = 1;          // Set all pins pull-down to enabled
PAC5XXX_GPIOA->OUTEN.b5 = 0;      // Set all pins to input
state = PAC5XXX_GPIOA->IN.b;      // Sample all pins input state
```

[SDK]

```
pac5xxx_gpio_out_pull_down_b(0);    // sets all pins in port with this mask
pac5xxx_gpio_out_pull_up_b(0xFF);   // sets all pins in port with this mask
pac5xxx_gpio_out_enable_b(0);       // sets all pins in port with this mask
```


CHANGING TO IO MODE FROM PERIPHERAL MODE

In the PAC52XX, any available GPIOA, GPIOB, GPIOD and GPIOE pin may be configured to be used for a peripheral or for IO. For information on how to set any port and pin to peripheral mode, see the PAC52XX User Guide or the Application Note on that peripheral. Below it shows how to change the ports and pins from peripheral mode to IO mode. Note that GPIOC does not have this feature.

To perform the following actions, the code below may be used.

Set GPIOA, pin 5 to IO mode as an output:

[Peripheral]

```
PAC5XXX_GPIOA->OUT.P5 = 0;        // Set to known state
PAC5XXX_GPIOA->OUTEN.P5 = 1;      // Set pin to digital output
PAC5XXX_GPIOA->PSEL.P5 = 0;       // Set pin to IO mode;
```

[SDK]

```
pac5xxx_gpio_out_set_a(0);           // sets all pins in port with this mask
pac5xxx_gpio_out_enable_a(0x20);     // sets all pins in port with this mask
pac5xxx_gpio_peripheral_select_a(0);// sets all pins in port with this mask
```

Set GPIOB, all pins to IO mode as an input:

[Peripheral]

```
PAC5XXX_GPIOB->OUTEN.b = 0;             // Set pins to digital input
PAC5XXX_GPIOB->PSEL.b = 0;              // Set pin to IO mode
```

[SDK]

```
pac5xxx_gpio_out_enable_b(0);        // sets all pins in port with this mask
pac5xxx_gpio_peripheral_select_b(0);// sets all pins in port with this mask
```

PAC52XX GPIO INTERRUPTS

In the PAC52XX, any available GPIO pin may be configured to be used to generate interrupts to the Cortex M0. The GPIO interrupt features allow the user to configure the following:

- Interrupt enable/disable (at GPIO peripheral)
- Interrupt enable/disable (at Cortex-M0 NVIC)
- Global Interrupt enable/disable (for Cortex-M0)
- Interrupt mask enable/disable
- Interrupt polarity
- Interrupt flag per pin

Because of the flexibility of the interrupt configuration, a few different registers must be written in a certain order, so it is recommended to use the SDK functions to configure interrupts. The SDK allows the user to configure the interrupt all at one time, or change individual interrupt features one at a time using the SDK functions.

Configuring Interrupts

To configure interrupts for a GPIO pin (or pins), the SDK allows the user to complete this by calling a single function for each GPIO port. These functions configure the interrupt pins, polarity and clear the interrupt flags in one function so the user does not have to perform these steps individually.

To perform this operation, the following code may be used.

Configure GPIOA, pin 5 for interrupts with high to low edge transition:

```
pac5xxx_gpio_configure_interrupt_b(0x20, 0x00); // arg: pin_mask, act_high_mask
```

This single function will do the following:

- Set the interrupt mask (based on the first argument)
- Set the interrupt polarity (low to high [0] or high to low [1])
- Enable the GPIO pin interrupt
- Enable the NVIC interrupt for GPIO port to the Cortex-M0
- Clear the interrupt flag (for any interrupts that have already fired)
- Un-mask the interrupt

The user must make sure that the global interrupt enable flag has already been enabled. This can be done through:

```
__enable_irq();   // Global interrupt enable
```

To disable global interrupts, the user should disable the global flag via:

```
__disable_irq();  // Global interrupt disable
```

Once these steps are performed, the user should be able to receive interrupts in the interrupt handler, as shown in the sections below.

Configuring Interrupts Manually

The PAC52XX provides functions that allow the user to configure the interrupts manually, without calling the function shown above. It is important that the functions be called in this order, to have them properly work. Specifically, the interrupt must be masked first, then the interrupt enabled, then the interrupt un-masked.

Below is the code that will properly enable the interrupts on GPIOB0, for high-to-low transitions:

```
pac5xxx_gpio_int_mask_b(0x01);                  // Set interrupt mask at startup (interrupt startup workaround)
pac5xxx_gpio_int_polarity_b(0);                 // Set interrupt polarity for high-to-low transitions
pac5xxx_gpio_int_enable_b(0x01);                // Enable interrupts in GPIO controller for GPIOB, pin 0
NVIC_EnableIRQ(GpioB_IRQn);                     // Enable interrupts in the global NVIC (Cortex-M0)
pac5xxx_gpio_int_flag_clear_b(0x01);            // Clear any active interrupt flags on GPIOB, pin 0 (write 1 to clear)
pac5xxx_gpio_int_mask_b(0);                     // Un-mask GPIOB, pin 0 and interrupts are now enabled
```

Once these steps are followed – assuming the global interrupt enable flag is set – interrupts will be processed by the Cortex-M0.

Changing Interrupt Polarity

The PAC52XX may configure interrupts for low-to-high or high-to-low transitions, but not both at the same time. If the user needs to configure the polarity of the interrupts, they may call the following functions.

Configuring High-to-Low Transitions on GPIOA, pin 5:

pac5xxx_gpio_int_polarity_a(0);

Configuring Low-to-High Transitions on GPIOB, pins 1-7:

pac5xxx_gpio_int_polarity_a(0x7F);

These functions may be called at any time, including when the interrupt is enabled and un-masked. If the function is called when interrupts are enabled and un-masked, then the next time an interrupt condition happens, the interrupt will be received. The interrupt does not have to be disabled and re-enabled for this.

For GPIOC, users should make sure that the input enable signal is set to 1, so that the GPIOC pins may receive digital input.

Processing Interrupts

Once the steps above are followed to process interrupts, interrupts can be received by the Cortex-M0.

Each GPIO port has its own interrupt handler that must be named as follows in the table below.

GPIO Port	Port Pin	Interrupt handler function signature
GPIOA	0-7	void GpioA_IRQHandler(void)
GPIOB	0-7	void GpioB_IRQHandler(void)
GPIOC	0-7	void GpioC_IRQHandler(void)
GPIOD	0-7	void GpioD_IRQHandler(void)
GPIOE	0-7	void GpioE_IRQHandler(void)

When an interrupt is generated, the interrupt flag is set for the pin. The interrupt flag is set whether the interrupt is enabled or not. Of course, the interrupt to the Cortex-M0 is only generated if the interrupt is enabled and unmasked.

Once the flag is set, the Cortex-M0 will call the interrupt service routine (ISR) for the given GPIO port. The user must define a function with the name of the ISR from the table above, or the default ISR will be called (in the system startup files and does nothing).

For example, if GPIOA, pin 0 has been configured to enable interrupts on a high-to-low edge transition as shown above then the following ISR would be defined to handle the interrupt:

```
void GpioA_IRQHandler(void)
{
  // handle interrupt

  led_on_red();

  // When complete, clear the interrupt flag

  pac5xxx_gpio_int_flag_clear_a(0x01);     // Clear interrupt flag on GPIOA, pin 0


}
```

Clearing the interrupt flag before the end of the ISR is important, because if the flag stays set, this ISR will be called repeatedly (according to priority) until the flag is cleared.

For more information on the Cortex-M0 interrupt configuration (interrupt priorities and scheduling, etc.) see the ARM Cortex-M0 documentation.

ABOUT ACTIVE-SEMI

Active-Semi, Inc. headquartered in Dallas, TX is a leading innovative semiconductor company with proven power management, analog and mixed-signal products for end-applications that require power conversion (AC/DC, DC/DC, DC/AC, PFC, etc.), motor drivers and control and LED drivers and control along with ARM microcontroller for system development.

Active-Semi's latest family of Power Application Controller (PAC)™ ICs offer high-level of integration with 32-bit ARM Cortex M0, along with configurable power management peripherals, configurable analog front-end with high-precision, high-speed data converters, single-ended and differential PGAs, integrated low-voltage and high-voltage gate drives. PAC IC offers unprecedented flexibility and ease in the systems design of various end-applications such as Wireless Power Transmitters, Motor drives, UPS, Solar Inverters and LED lighting, etc. that require a microcontroller, power conversion, analog sensing, high-voltage gate drives, open-drain outputs, analog & digital general purpose IO, as well as support for wired and wireless communication. More information and samples can be obtained from http://www.active-semi.com or by emailing marketing@active-semi.com

Active-Semi shipped its 1 Billionth IC in 2012, and has over 120 in patents awarded and pending approval.