

# *Embedded 'C' for Zynq*

---

*Cristian Sisterna*

*Universidad Nacional de San Juan*

*Argentina*

# Embedded C

---

## Embedded C

---

From Wikipedia, the free encyclopedia

**Embedded C** is a set of language extensions for the C Programming language by the C Standards committee to address commonality issues that exist between C extensions for different embedded systems. Historically, embedded C programming requires nonstandard extensions to the C language in order to support exotic features such as fixed-point arithmetic, multiple distinct memory banks, and basic I/O operations.

In 2008, the C Standards Committee extended the C language to address these issues by providing a common standard for all implementations to adhere to. It includes a number of features not available in normal C, such as, fixed-point arithmetic, named address spaces, and basic I/O hardware addressing.

# Difference Between C and *Embedded C*

---

Embedded systems programming is different from developing applications on a desktop computers. Key characteristics of an embedded system, when compared to PCs, are as follows:

- ❑ Embedded devices have resource constraints(limited ROM, limited RAM, limited stack space, less processing power)
- ❑ Components used in embedded system and PCs are different; embedded systems typically uses smaller, less power consuming components
- ❑ Embedded systems are more tied to the hardware
- ❑ Two salient features of Embedded Programming are ***code speed*** and ***code size***. Code speed is governed by the processing power, timing constraints, whereas code size is governed by available program memory and use of programming language.

# Difference Between C and Embedded C

---

Though **C** and **Embedded C** appear different and are used in different contexts, they have more similarities than the differences. Most of the constructs are same; the difference lies in their applications.

**C** is used for desktop computers, while **Embedded C** is for microcontroller based applications.

Compilers for **C** (ANSI C) typically generate OS dependent executables. **Embedded C** requires compilers to create files to be downloaded to the microcontrollers/microprocessors where it needs to run. Embedded compilers give access to all resources which is not provided in compilers for desktop computer applications.

Embedded systems often have the real-time constraints, which is usually not there with desktop computer applications.

Embedded systems often do not have a console, which is available in case of desktop applications.

# Advantages of Using *Embedded C*

---

- It is small and reasonably simpler to learn, understand, program and debug
- C Compilers are available for almost all embedded devices in use today, and there is a large pool of experienced C programmers
- Unlike assembly, C has advantage of processor-independence and is not specific to any particular microprocessor/ microcontroller or any system. This makes it convenient for a user to develop programs that can run on most of the systems
- As C combines functionality of assembly language and features of high level languages, C is treated as a 'middle-level computer language' or 'high level assembly language'
- It is fairly efficient
- It supports access to I/O and provides ease of management of large embedded projects
- Objected oriented language, C++ is not apt for developing efficient programs in resource constrained environments like embedded devices.

---

# Reviewing Embedded 'C' Basic Concepts

---

# Basic Data Types

Type	Size	Unsigned Range	Signed Range
char	8 bits	0 to 255	−128 to 127
short int	8 bits	0 to 255	−128 to 127
int	16 bits	0 to 65535	−32768 to 32767
long int	32 bits	0 to 4294967295	−2147483648 to 2147483647

```
typedef unsigned char UINT8;  
typedef signed char SINT8;  
typedef unsigned int UINT16;  
typedef int SINT16;  
typedef unsigned long int UINT32;  
typedef long int SINT32;
```

# 'SDK' Basic Data Types

---

xbasic\_types.h

```
typedef unsigned char   Xuint8;    /**< unsigned 8-bit */
typedef char           Xint8;     /**< signed 8-bit */
typedef unsigned short Xuint16;   /**< unsigned 16-bit */
typedef short          Xint16;    /**< signed 16-bit */
typedef unsigned long  Xuint32;   /**< unsigned 32-bit */
typedef long           Xint32;    /**< signed 32-bit */
typedef float          Xfloat32;  /**< 32-bit floating point */
typedef double         Xfloat64;  /**< 64-bit double precision FP */
typedef unsigned long  Xboolean;  /**< boolean (XTRUE or XFALSE) */
```

xil\_types.h

```
typedef uint8_t  u8;
typedef uint16_t u16;
typedef uint32_t u32;
```



# Local vs Global Variables

---

Variables in C can be classified by their scope

## Local Variables

Accessible only by the function within which they are declared and are allocated storage on *the stack*

The '***static***' access modifier causes that the local variable to be permanently allocated storage in memory, like a global variable

## Global Variables

Accessible by any part of the program and are allocated permanent storage in RAM



Returning a pointer to a GLOBAL or STATIC variable is quite safe



# Local Variables

---

- ❖ The 'static' access modifier causes that the local variable to be permanently allocated storage in memory, like a global variable, so the value is preserved between function calls (but still is local)
- ❖ Local variables only occupy RAM while the function to which they belong is running
- ❖ Usually the stack pointer addressing mode is used (This addressing mode requires one extra byte and one extra cycle to access a variable compared to the same instruction in indexed addressing mode)
  - ❖ If the code requires several consecutive accesses to local variables, the compiler will usually transfer the stack pointer to the 16-bit index register and use indexed addressing instead

# Global Variables

---

- ❖ *Global variables* are allocated permanent storage in memory at an absolute address determined when the code is linked
- ❖ The memory occupied by a *global variable* cannot be reused by any other variable
- ❖ *Global variables* are not protected in any way, so any part of the program can access a global variable at any time
  - ❖ This means that the variable data could be corrupted if part of the variable is derived from one value and the rest of the variable is derived from another value
- ❖ The 'static' access modifier may also be used with *global variables*
  - ❖ This gives some degree of protection to the variable as it restricts access to the variable to those functions in the file in which the variable is declared
- ❖ The compiler will generally use the extended addressing mode to access *global variables* or indexed addressing mode if they are accessed through a pointer

# Other Application for the 'static' modifier

---

By default, all functions and variables declared *in global space* have external linkage and are visible to the entire program. Sometimes you require global variables or functions that have internal linkage: they should be visible within a single compilation unit, but not outside. Use the `static` keyword to restrict the scope of variables.

```
#include "xparameters.h"
#include "xgpio.h"
#include "xgpiops.h"

static XGpioPs psGpioInstancePtr;
static int iPinNumber = 7; /*Led LD9 is connected to MIO pin 7*/
//=====

int main (void)
{
    XGpio sw, led;
    int i, pshb_check, sw_check;
    static XGpio GPIOInstance_Ptr;
```

# Volatile Variable

---

The value of *volatile variables* may change from outside the program. For example, you may wish to read an A/D converter or a port whose value is changing. *Often your compiler may eliminate code to read the port as part of the compiler's code optimization process* if it does not realize that some outside process is changing the port's value. You can avoid this by declaring the variable `volatile`.

```
#define MYPORT 0xDEADB33F

volatile char *portptr = (char*)MYPORT;
*portptr = 'A';
*portptr = 'B';
```

Without "*volatile*", the first write may be optimized out

# Volatile Variable

```
1  #include <stdio.h>
2
3  /* Optimization code snippet 1 */
4  #include<stdio.h>
5
6  int x = 0;
7
8  int main()
9  {
10     if (x == 0) // This condition is always
11     {
12         printf(" x = 0 \n");
13     }
14     else        // Else part will be optimized
15     {
16         printf(" x != 0 \n");
17     }
18     return 0;
19 }
```

```
1  #include<stdio.h>
2
3  volatile int x = 0;    /* volatile Keyword*/
4
5  int main()
6  {
7     x = 0;
8
9     if (x == 0)
10    {
11        printf(" x = 0 \n");
12    }
13    else        // Now compiler never optimizes else part because the
14    {            // variable is declared as volatile
15        printf(" x != 0 \n");
16    }
17    return 0;
18 }
```

# Functions Data Types

---

A function data type defines the value that a subroutine can return

- ❖ A function of type `int` returns a signed integer value
- ❖ Without a specific return type, any function returns an `int`
- ❖ To avoid confusion, you should always declare `main()` with return type `void`

```
void XGpioPs_IntrEnable(XGpioPs *InstancePtr, u8 Bank, u32 Mask);  
void XGpioPs_IntrDisable(XGpioPs *InstancePtr, u8 Bank, u32 Mask);  
u32 XGpioPs_IntrGetEnabled(XGpioPs *InstancePtr, u8 Bank);  
u32 XGpioPs_IntrGetStatus(XGpioPs *InstancePtr, u8 Bank);
```

# Parameters Data Types

Indicate the values to be passed into the function and the memory to be reserved for storing them

```
int XGpio_Initialize(XGpio *InstancePtr, u16 DeviceId);  
XGpio /**  
/* Initialize the XGpio instance provided by the caller based on the  
/* given DeviceID.  
/* A  
/*  
/* Nothing is done except to initialize the InstancePtr.  
/*  
/* @param InstancePtr is a pointer to an XGpio instance. The memory the  
/* pointer references must be pre-allocated by the caller. Further  
/* calls to manipulate the instance/driver through the XGpio API  
/* must be made with this pointer.  
/* @param DeviceId is the unique id of the device controlled by this XGpio  
/* instance. Passing in a device id associates the generic XGpio
```



# Structures

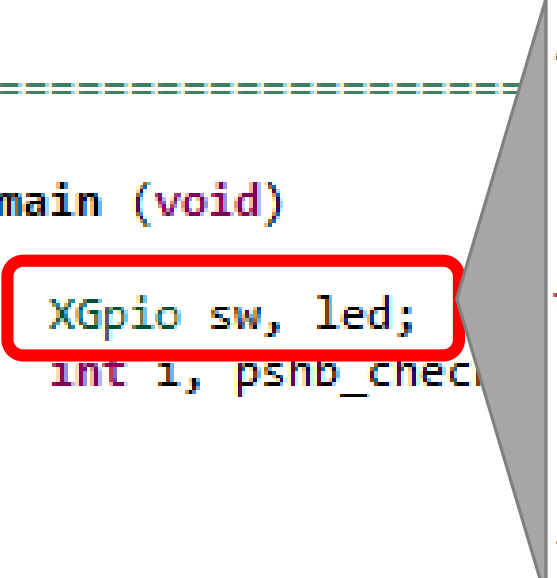
```
#include "xparameters.h"
#include "xgpio.h"
#include "xgpiops.h"

static XGpioPs psGpioInstancePtr;
static int iPinNumber = 7; /*Led LD9

/**
//===== * The XGpio driver instance data. The user is required to allocate a
* variable of this type for every GPIO device in the system. A pointer
* to a variable of this type is then passed to the driver API functions.
*/

int main (void)
{
    XGpio sw, led;
    int i, psnb_check;

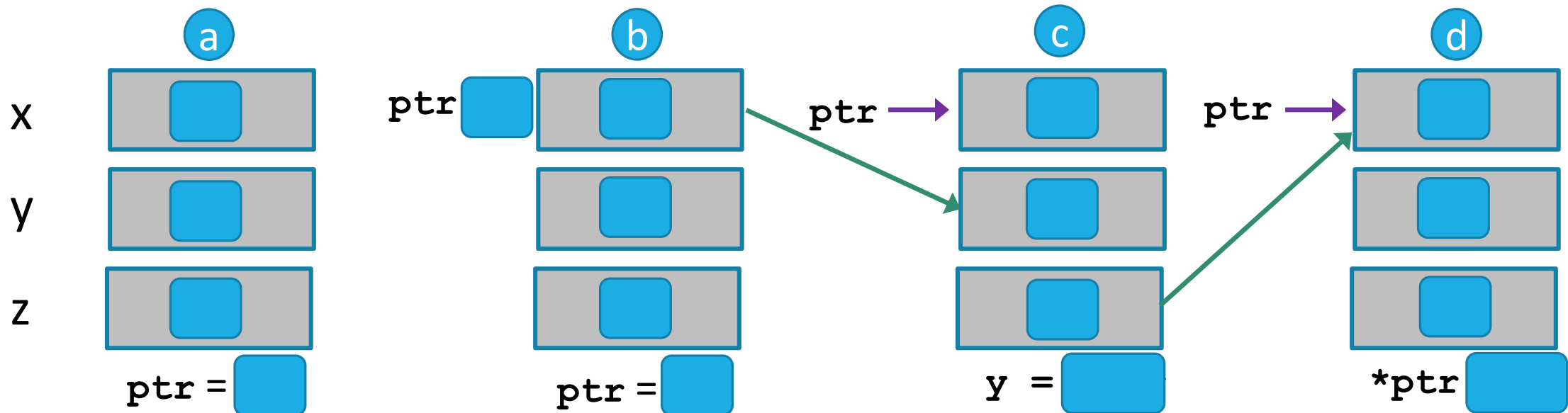
    typedef struct {
        u32 BaseAddress;      /* Device base address */
        u32 IsReady;          /* Device is initialized and ready */
        int InterruptPresent; /* Are interrupts supported in h/w */
        int IsDual;           /* Are 2 channels supported in h/w */
    } XGpio;
```



# Review of 'C' Pointer

In 'C', the pointer data type corresponds to a MEMORY ADDRESS

- a `int x = 1, y = 5, z = 8, *ptr;`
- b `ptr = &x; // ptr gets (point to) address of x`
- c `y = *ptr; // content of y gets content pointed by ptr`
- d `*ptr = z; // content pointed by ptr gets content of z`



---

# 'C' Techniques for low-level I/O Operations

---

# Bit Manipulation in 'C'

---

Bitwise operators in 'C': `~` (**not**), `&` (**and**), `|` (**or**), `^` (**xor**)  
which operate on one or two operands at bit levels

```
u8 mask = 0x60;      //0110_0000 mask bits 6 and 5
u8 data = 0xb3        //1011_0011 data
u8 d0, d1, d2, d3;    //data to work with in the coming example
. . .
```

```
d0 = data & mask;     // 0010_0000; isolate bits 6 and 5 from data
d1 = data & ~mask;    // 1001_0011; clear bits 6 and 5 of data
d2 = data | mask;     // 1111_0011; set bits 6 and 5 of data
d3 = data ^ mask;     // 1101_0011; toggle bits 6 and 5 of data
```

# Bit Shift Operators

---

Both operands of a bit shift operator must be integer values

The **right shift operator** shifts the data right by the specified number of positions. Bits shifted out the right side disappear. With unsigned integer values, 0s are shifted in at the high end, as necessary. For signed types, the values shifted in is implementation-dependant. The binary number is shifted right by *number* bits.

```
x >> number;
```

The **left shift operator** shifts the data right by the specified number of positions. Bits shifted out the left side disappear and new bits coming in are 0s. The binary number is shifted left by *number* bits

```
x << number;
```

# Bit Shift Example

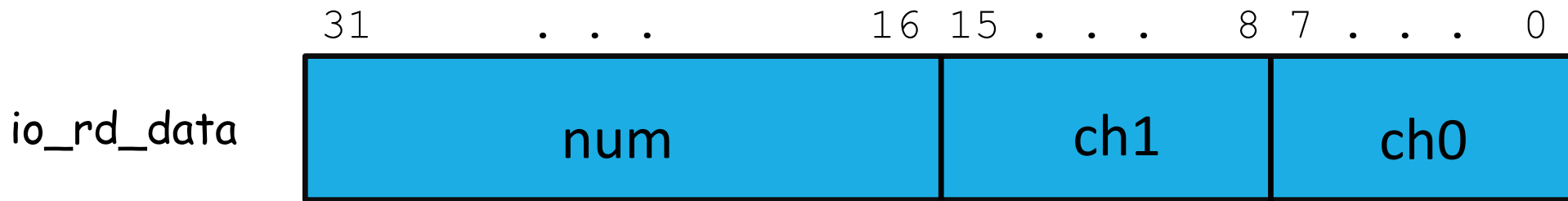
---

```
void led_knight_rider(XGpio *pLED_GPIO, int nNumberOfTimes)
{
    int i=0;        int j=0;
    u8 uchLedStatus=0;
    // Blink the LEDs back and forth nNumberOfTimes
    for (i=0; i<nNumberOfTimes; i++)
    {
        for (j=0; j<8; j++) // Scroll the LEDs up
        {
            uchLedStatus = 1 << j;
            XGpio_DiscreteWrite(pLED_GPIO, 1, uchLedStatus);
            delay(ABOUT_ONE_SECOND / 15);
        }
        for (j=0; j<8; j++) // Scroll the LEDs down
        {
            uchLedStatus = 8 >> j;
            XGpio_DiscreteWrite(pLED_GPIO, 1, uchLedStatus);
            delay(ABOUT_ONE_SECOND / 15);
        }
    }
}
```

# Unpacking Data

There are cases that in the same memory address different fields are stored

Example: let's assume that a 32-bit memory address contains a 16-bit field for an integer data and two 8-bit fields for two characters



```
u32 io_rd_data;  
int num;  
char ch1, ch0;
```

Unpacking {

```
io_rd_data = my_iord(...);  
num = (int) ((io_rd_data & 0xffff0000) >> 16);  
ch1 = (char) ((io_rd_data & 0x0000ff00) >> 8);  
ch0 = (char) ((io_rd_data & 0x000000ff));
```

# Packing Data

There are cases that in the same memory address different fields are written

Example: let's assume that a 32-bit memory address will be written as a 16-bit field for an integer data and two 8-bit fields for two characters



```
u32 wr_data;  
int num = 5;  
char ch1, ch0;
```

Packing {

```
wr_data = (u32) (num); //num[15:0]  
wr_data = (wr_data << 8) | (u32) ch1; //num[23:8], ch1[7:0]  
wr_data = (wr_data << 8) | (u32) ch0; //num[31:16], ch1[15:8]  
my_iowr( . . . , wr_data) ; //ch0[7:0]
```



# I/O Read Macro

## Read from an Input

```
int switch_s1;  
.  
.  
switch_s1 = *(volatile int *) (0x00011000);
```

```
#define SWITCH_S1_BASE = 0x00011000;  
.  
switch_s1 = *(volatile int *) (SWITCH_S1_BASE);
```

```
#define SWITCH_S1_BASE = 0x00011000;  
#define my_iord(addr) (*(volatile int *) (addr))  
.  
switch_s1 = my_iord(SWITCH_S1_BASE);    //
```

Macro

# I/O Write Macro

## Write to an Output

```
char pattern = 0x01;  
.  
.  
.  
*(0x11000110) = pattern;
```

```
#define LED_L1_BASE = 0x11000110;  
.  
.  
.  
*(LED_L1_BASE) = pattern;
```

```
#define LED_L1_BASE = 0x11000110;  
#define my_iowr(addr, data)  (*(int *) (addr) = (data))  
.  
.  
.  
my_iowr(LED_L1_BASE, (int)pattern);    //
```

Macro

---

# Basic 'C' Program Template

---

# Basic Embedded Program Architecture

---

An embedded application consists of a collection tasks, implemented by hardware accelerators, software routines, or both.

```
#include "nnnnn.h"
#include <ppppp.h>
main ()
{
    sys_init(); //
    while(1) {
        task_1();
        task_2();
        . . .
        task_n();
    }
}
```

# Basic Example

---

The flashing-LED system turns on and off *two* LEDs alternatively according to the interval specified by the *ten* sliding switches

Tasks ????



1. reading the interval value from the switches
2. toggling the two LEDs after a specific amount of time

# Basic Example

---

```
main ()
{
while (1) {
    . . .
    task_1 ();
    task_2 ();
    . . .
}
}
```

```
#include "nnnnn.h"
#include "aaaaa.h"
```

```
main ()
{
int period;

while (1) {
    read_sw (SWITCH_S1_BASE, &period);
    led_flash (LED_L1_BASE, period);
}
}
```

# Basic Example - Reading

```
/* **** */
* function: read_sw ()
* purpose: get flashing period from switches
* argument:
*     sw-base: base address of switch PIO
*     period: pointer to period
* return:
*     updated period
* note :
* **** */
void read_sw(u32 switch_base, int *period)
{
    *period = my_iord(switch_base) & 0x000000ff; //read flashing period
                                                    // from switch
}
```

# Basic Example - Writing

```

/*****
* function: led.flash ()
* purpose: toggle 2 LEDs according to the given period
* argument:
*         led-base: base address of discrete LED PIO
*         period: flashing period in ms
* return :
* note :
* - The delay is done by estimating execution time of a dummy for loop
* - Assumption: 400 ns per loop iteration (2500 iterations per ms)
* - 2 instruct. per loop iteration /10 clock cycles per instruction /20ns per clock cycle(50-MHz clock)
*****/
void led_flash(u32 addr_led_base, int period)
{
    static u8 led_pattern = 0x01;           // initial pattern
    unsigned long i, itr;
    led_pattern ^= 0x03;                     // toggle 2 LEDs (2 LSBs)
    my_iowr(addr_led_base, led_pattern);    // write LEDs
    itr = period * 2500;
    for (i=0; i<itr; i++) {}                // dummy loop for delay
}

```



# Basic Example – Read / Write

```
main()
{
    int period;

    while(1){
        read_sw(SWITCH_S1_BASE, &period);
        led_flash(LED_L1_BASE, period);
    }
}
```

```
void read_sw(u32 switch_base, int *period)
{
    *period = my_iord(switch_base) & 0x000003ff;
}
```

```
void led_flash(u32 addr_led_base, int period)
{
    static u8 led_pattern = 0x01;
    unsigned long i, itr;
    led_pattern ^= 0x03;
    my_iowr(addr_led_base, led_pattern);
    itr = period * 2500;
    for (i=0; i<itr; i++) {}
}
```

---

# Read/Write From/To GPIO Inputs and Outputs

---

# Steps for Reading from a GPIO

---

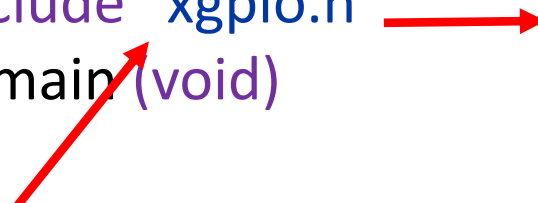
1. Create a GPIO instance
2. Initialize the GPIO
3. Set data direction
4. Read the data

# Steps for Reading from a GPIO – Step 1

## 1. Create a GPIO instance

```
#include "xparameters.h"
#include "xgpio.h"
int main(void)
{
    XGpio switches;
    XGpio leds;
    ...
}

/**
 * The XGpio driver instance data. The user is required to allocate a
 * variable of this type for every GPIO device in the system. A pointer
 * to a variable of this type is then passed to the driver API functions.
 */
typedef struct {
    u32 BaseAddress;          /* Device base address */
    u32 IsReady;              /* Device is initialized and ready */
    int InterruptPresent;     /* Are interrupts supported in h/w */
    int IsDual;               /* Are 2 channels supported in h/w */
} XGpio;
```



The **XGpio** driver instance data. The user is required to allocate a *variable of this type* for *every GPIO device in the system*. A pointer to a variable of this type is then passed to the driver API functions.

# Steps for Reading from a GPIO – Step 2

---

## 2. Initialize the GPIO

```
(int) XGpio_Initialize(XGpio *InstancePtr, u16 DeviceID);
```

**InstancePtr:** is a pointer to an XGpio instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the XGpio API must be made with this pointer.

**DeviceID:** is the unique id of the device controlled by this XGpio component. Passing in a device ID associates the generic XGpio instance to a specific device, as chosen by the caller or application developer.

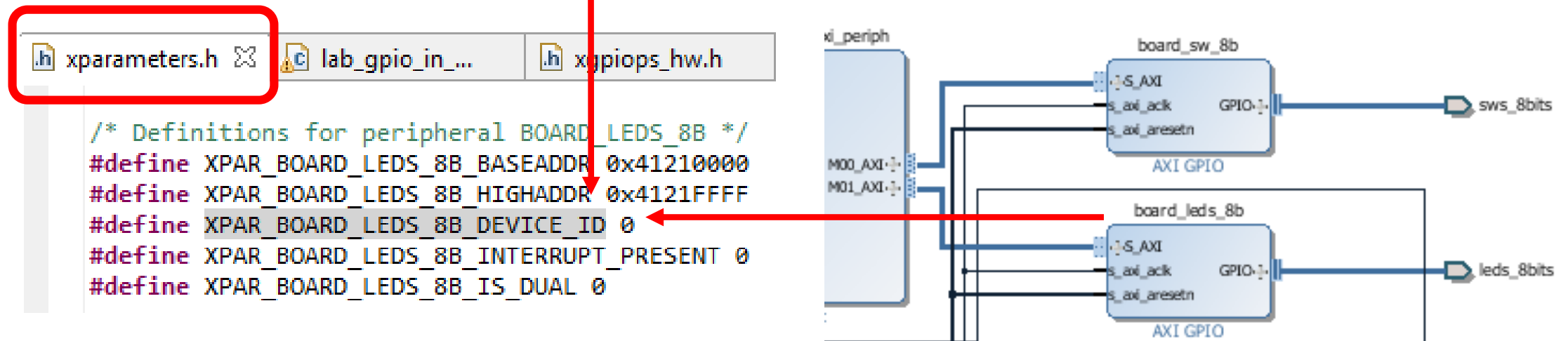
### @return

- XST\_SUCCESS if the initialization was successful.
  - XST\_DEVICE\_NOT\_FOUND if the device configuration data was not
- } xstatus.h

# Steps for Reading from a GPIO – Step 2(cont')

```
(int) XGpio_Initialize(XGpio *InstancePtr, u16 DeviceID);
```

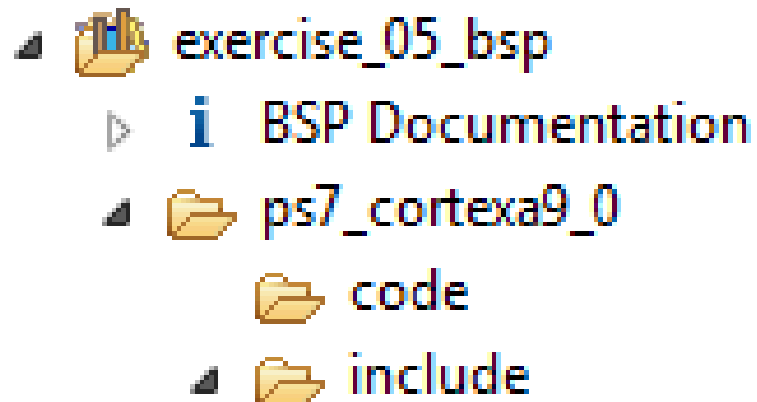
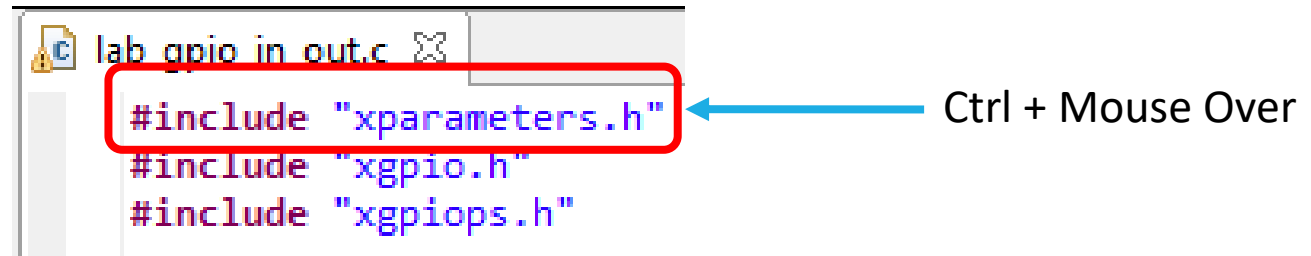
```
// AXI GPIO switches initialization  
XGpio_Initialize (&switches, XPAR_BOARD_SW_8B_DEVICE_ID);  
// AXI GPIO leds initialization  
XGpio_Initialize (&led, XPAR_BOARD_LEDS_8B_DEVICE_ID);
```



# *xparameters.h*

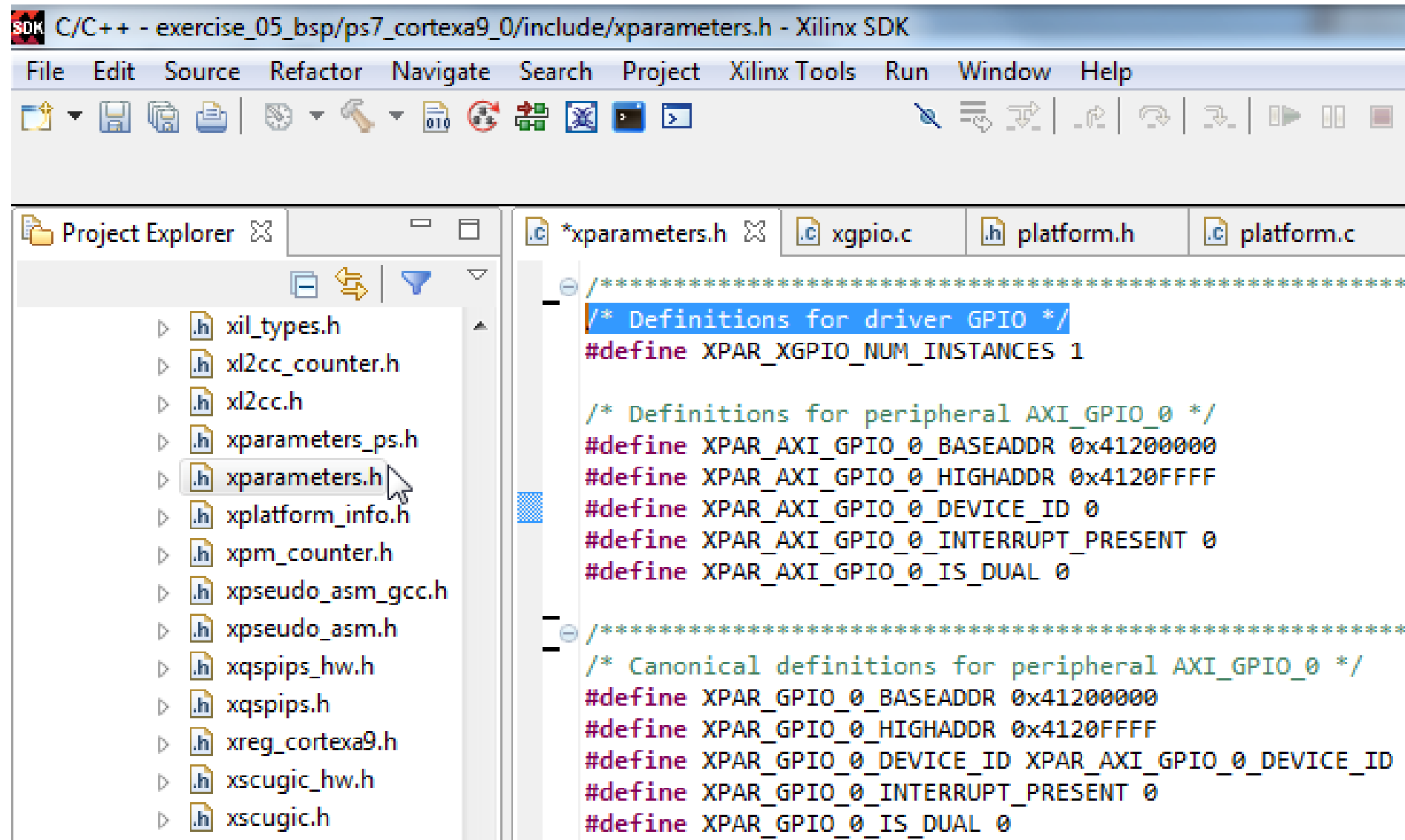
The *xparameters.h* file contains the address map for peripherals in the created system

This file is generated from the hardware platform description from Vivado



*xparameters.h* file can be found underneath the include folder in the ps7\_cortexa9\_0 folder of the BSP main folder

# xparameters.h



The screenshot shows the Xilinx SDK IDE with the file `xparameters.h` open. The Project Explorer on the left lists various header files, with `xparameters.h` selected. The main editor window displays the content of `xparameters.h`, which includes definitions for GPIO instances and peripheral AXI GPIO 0.

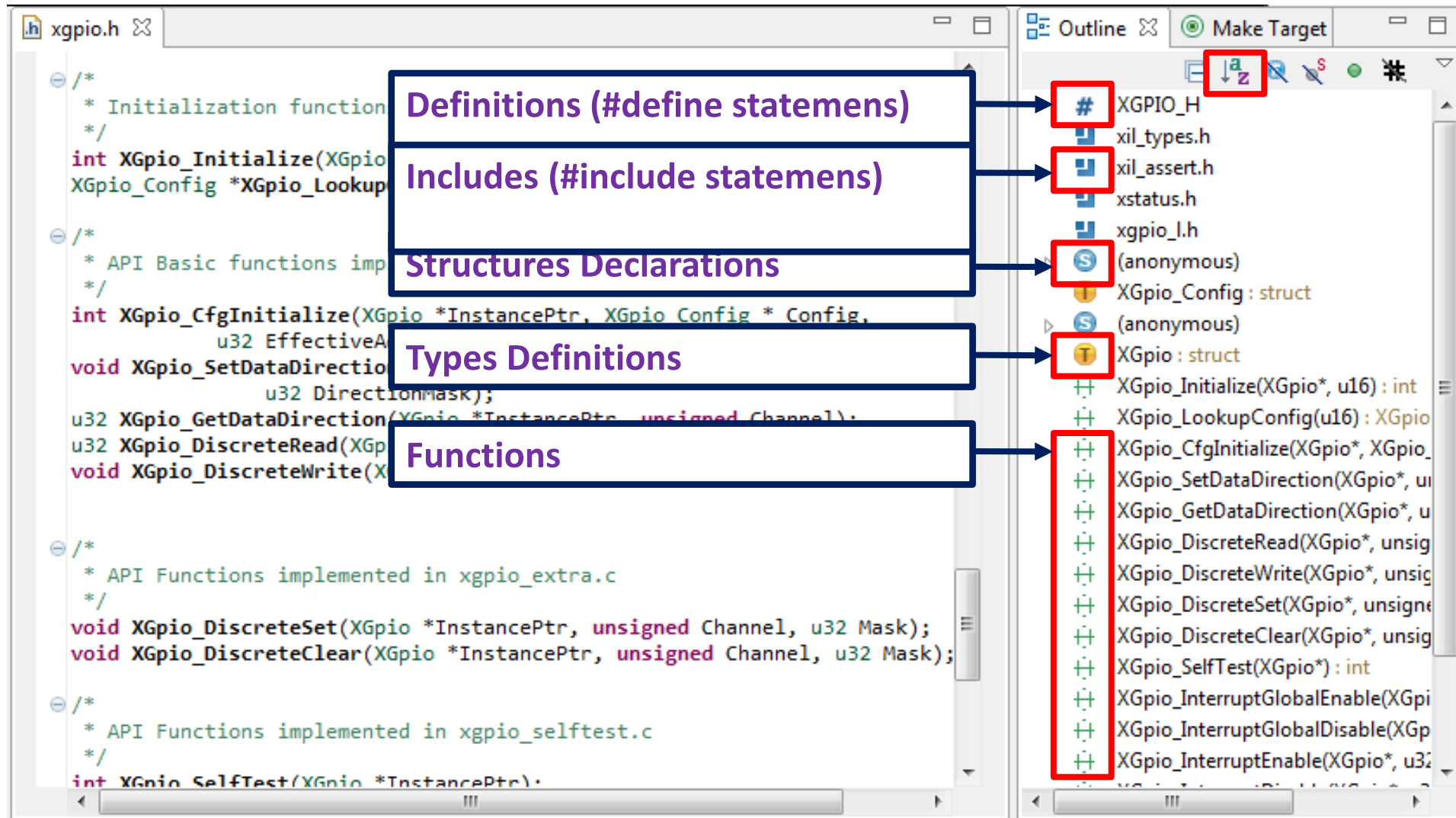
```
SDK C/C++ - exercise_05_bsp/ps7_cortexa9_0/include/xparameters.h - Xilinx SDK
File Edit Source Refactor Navigate Search Project Xilinx Tools Run Window Help
Project Explorer
  .h xil_types.h
  .h xl2cc_counter.h
  .h xl2cc.h
  .h xparameters_ps.h
  .h xparameters.h
  .h xplatform_info.h
  .h xpm_counter.h
  .h xpseudo_asm_gcc.h
  .h xpseudo_asm.h
  .h xqspips_hw.h
  .h xqspips.h
  .h xreg_cortexa9.h
  .h xscugic_hw.h
  .h xscugic.h
*xparameters.h xgpio.c platform.h platform.c
/* *****
/* Definitions for driver GPIO */
#define XPAR_XGPIO_NUM_INSTANCES 1

/* Definitions for peripheral AXI_GPIO_0 */
#define XPAR_AXI_GPIO_0_BASEADDR 0x41200000
#define XPAR_AXI_GPIO_0_HIGHADDR 0x4120FFFF
#define XPAR_AXI_GPIO_0_DEVICE_ID 0
#define XPAR_AXI_GPIO_0_INTERRUPT_PRESENT 0
#define XPAR_AXI_GPIO_0_IS_DUAL 0

/* *****
/* Canonical definitions for peripheral AXI_GPIO_0 */
#define XPAR_GPIO_0_BASEADDR 0x41200000
#define XPAR_GPIO_0_HIGHADDR 0x4120FFFF
#define XPAR_GPIO_0_DEVICE_ID XPAR_AXI_GPIO_0_DEVICE_ID
#define XPAR_GPIO_0_INTERRUPT_PRESENT 0
#define XPAR_GPIO_0_IS_DUAL 0
```



# xgpio.h – Outline Pane



# Steps for Reading from a GPIO - Step 3

---

## 3. Set data direction

```
void XGpio_SetDataDirection (XGpio *InstancePtr, unsigned Channel, u32 DirectionMask);
```

**InstancePtr:** is a pointer to an XGpio instance to be worked on.

**Channel:** contains the channel of the XGpio (1 or 2) to operate with.

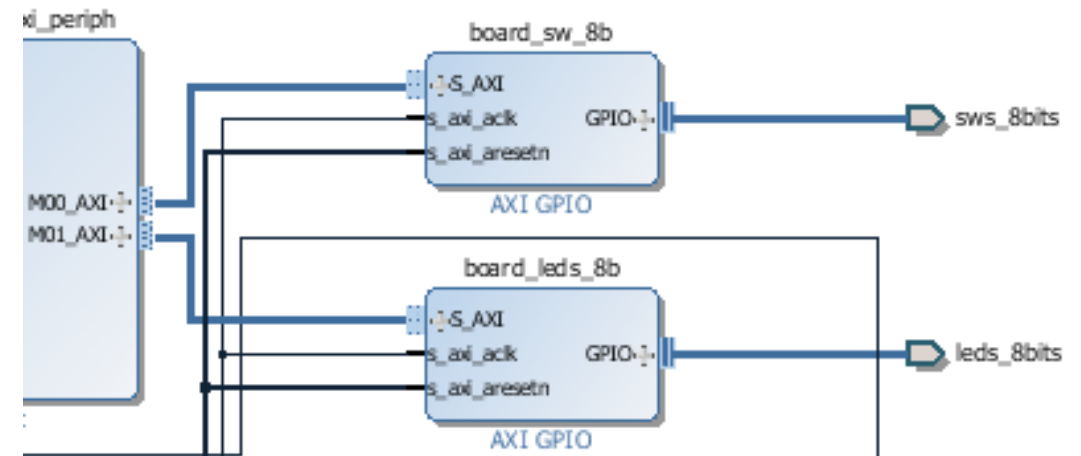
**DirectionMask:** is a bitmask specifying which bits are inputs and which are outputs. Bits set to '0' are output, bits set to '1' are inputs.

**Return:** none

# Steps for Reading from a GPIO - Step 3 (cont')

```
void XGpio_SetDataDirection (XGpio *InstancePtr, unsigned Channel, u32 DirectionMask);
```

```
// AXI GPIO switches: bits direction configuration  
XGpio_SetDataDirection(&switches, 1, 0xffffffff);
```



# Steps for Reading from a GPIO – Step 4

---

## 4. Read the data

```
u32 XGpio_DiscreteRead (XGpio *InstancePtr, unsigned Channel);
```

**InstancePtr:** is a pointer to an XGpio instance to be worked on.

**Channel:** contains the channel of the XGpio (1 o 2) to operate with.

**Return:** read data

# Steps for Reading from a GPIO – Step 4 (cont')

---

```
u32 XGpio_DiscreteRead (XGpio *InstancePtr, unsigned Channel);
```

```
// AXI GPIO: read data from the switches  
sw_check = XGpio_DiscreteRead(&switches, 1);
```

# Steps for Writing to GPIO

---

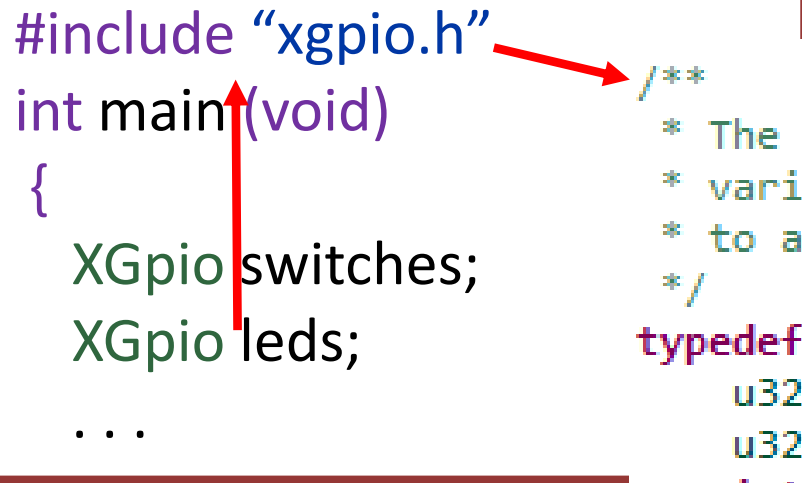
1. Create a GPIO instance
2. Initialize the GPIO
3. Read the data

# Steps for Writing to a GPIO – Step 1

---

## 1. Create a GPIO instance

```
#include "xgpio.h"
int main(void)
{
    XGpio switches;
    XGpio leds;
    ...
}
```



```
/**
 * The XGpio driver instance data. The user is required to allocate a
 * variable of this type for every GPIO device in the system. A pointer
 * to a variable of this type is then passed to the driver API functions.
 */
typedef struct {
    u32 BaseAddress;      /* Device base address */
    u32 IsReady;          /* Device is initialized and ready */
    int InterruptPresent; /* Are interrupts supported in h/w */
    int IsDual;           /* Are 2 channels supported in h/w */
} XGpio;
```

The **XGpio** driver instance data. The user is required to allocate a variable of this type for *every GPIO device in the system*. A pointer to a variable of this type is then passed to the driver API functions.

# Steps for Writing to a GPIO – Step 2

---

## 2. Initialize the GPIO

```
(int) XGpio_Initialize(XGpio *InstancePtr, u16 DeviceID);
```

**InstancePtr**: is a pointer to an XGpio instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the XGpio API must be made with this pointer.

**DeviceID**: is the unique id of the device controlled by this XGpio component. Passing in a device ID associates the generic XGpio instance to a specific device, as chosen by the caller or application developer.

### @return

- XST\_SUCCESS if the initialization was successful.
  - XST\_DEVICE\_NOT\_FOUND if the device configuration data was not
- } xstatus.h



# Steps for Writing to a GPIO – Step 2(cont')

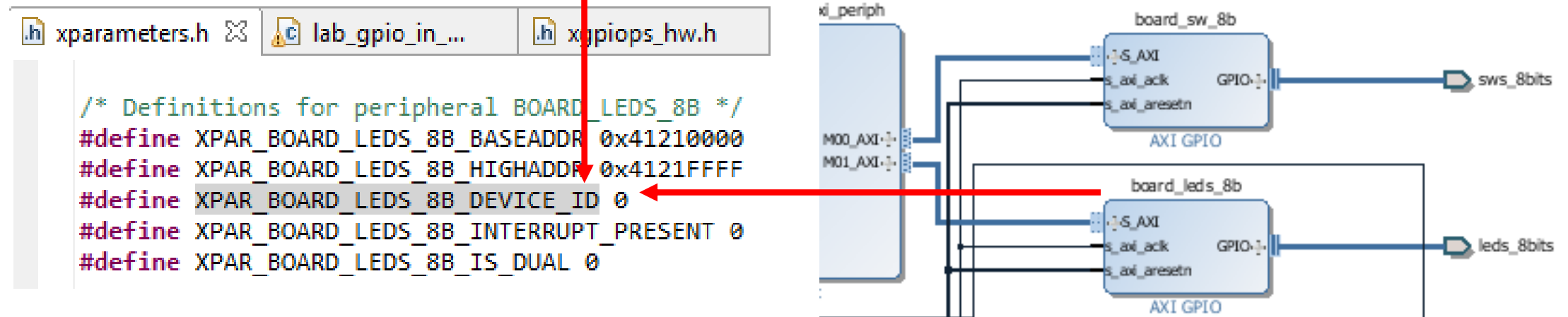
```
(int) XGpio_Initialize (XGpio *InstancePtr, u16 DeviceID);
```

```
// AXI GPIO switches initialization
```

```
XGpio_Initialize (&switches, XPAR_BOARD_SW_8B_DEVICE_ID);
```

```
// AXI GPIO leds initialization
```

```
XGpio_Initialize (&led, XPAR_BOARD_LEDS_8B_DEVICE_ID);
```



# Steps for Writing to a GPIO – Step 3

---

## 3. Write the data

```
void XGpio_DiscreteWrite (XGpio *InstancePtr, unsigned Channel, u32 Data);
```

**InstancePtr:** is a pointer to an XGpio instance to be worked on.

**Channel:** contains the channel of the XGpio (1 or 2) to operate with.

**Data:** Data is the value to be written to the discrete register

**Return:** none

# Steps for Writing to a GPIO – Step 3 (cont')

---

```
void XGpio_DiscreteWrite (XGpio *InstancePtr, unsigned Channel, u32 Data);
```

```
// AXI GPIO: read data from the switches  
sw_check = XGpio_DiscreteRead(&switches, 1);  
// AXI GPIO: write data (sw_check) to the LEDs  
XGpio_DiscreteWrite(&led, 1, sw_check);
```

---

# IP Drivers for Custom IP

---

# My VHDL Code for the Future IP

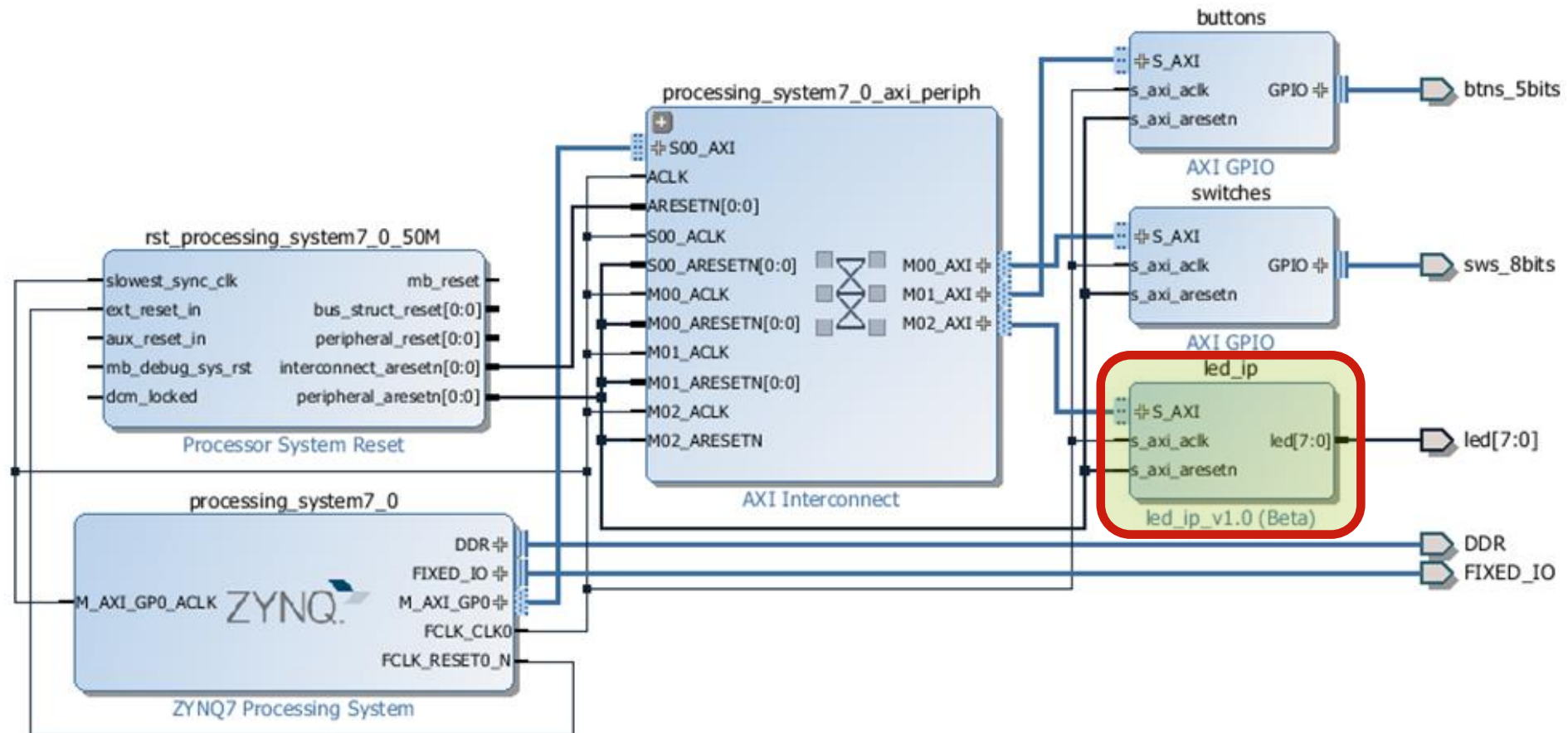
```
1 -----
2 -- lab name: lab_custom_ip
3 -- component name: my_led_ip
4 -- author: cas
5 -- version: 1.0
6 -- description: simple logic to
7 -----
8 library ieee;
9 use ieee.std_logic_1164.all;
10
11 entity lab_led_ip is
12
13     generic (
14         led_width : integer := 8);          -- 8 LEDs
15     port (
16         -- clock and reset
17         S_AXI_ACLK   : in std_logic;
18         S_AXI_ARESETN : in std_logic;
19         -- write data channel
20         S_AXI_WDATA  : in std_logic_vector(31 downto 0);
21         SLV_REG_WREN  : in std_logic;
22         -- address channel
23         AXI_AWADDR    : in std_logic_vector(3 downto 0);
24         -- my inputs / outputs --
25         -- output
26         LED           : out std_logic_vector(led_width-1 downto 0)
27     );
28 end entity lab_led_ip;
```

AXI4-Lite IP

```
30 architecture beh of lab_led_ip is
31
32 begin -- architecture beh
33
34     process(S_AXI_ACLK, S_AXI_ARESETN)
35     begin
36         if(S_AXI_ARESETN='0') then
37             LED <= (others=>'0');
38         elsif(rising_edge(S_AXI_ACLK)) then
39             if(SLV_REG_WREN='1' and AXI_AWADDR="0000") then
40                 LED <= S_AXI_WDATA(led_width-1 downto 0);
41             end if;
42         end if;
43     end process;
44 end architecture beh;
```

Address Decode & Write Enable

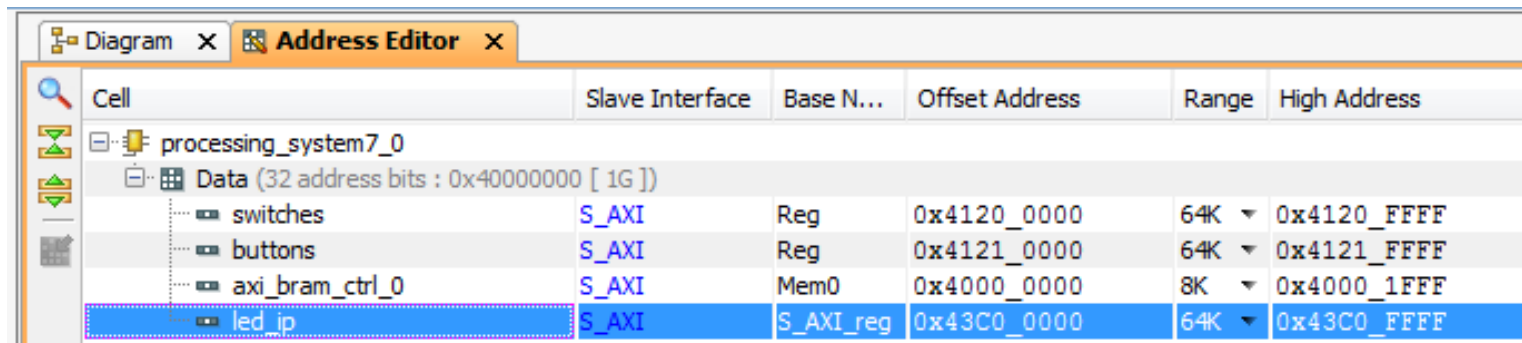
# Custom IP



# System Level Address Map

Address Range	CPU's and ACP	AXI_HP	Other Bus Masters <sup>(1)</sup>	Notes
0000_0000 to 0003_FFFF <sup>(2)</sup>	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU <sup>(3)</sup>
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see <a href="#">Table 4-6</a>
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see <a href="#">Table 4-5</a>
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see <a href="#">Table 4-3</a>
F800_1000 to F880_FFFF	PS		PS	PS System registers, see <a href="#">Table 4-7</a>
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see <a href="#">Table 4-4</a>
FC00_0000 to FDFD_FFFF <sup>(4)</sup>	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF <sup>(2)</sup>	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

# My IP – Memory Address Range



Cell	Slave Interface	Base N...	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [ 1G ])					
switches	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
buttons	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	8K	0x4000_1FFF
led_ip	S_AXI	S_AXI_reg	0x43C0_0000	64K	0x43C0_FFFF



# Custom IP Drivers

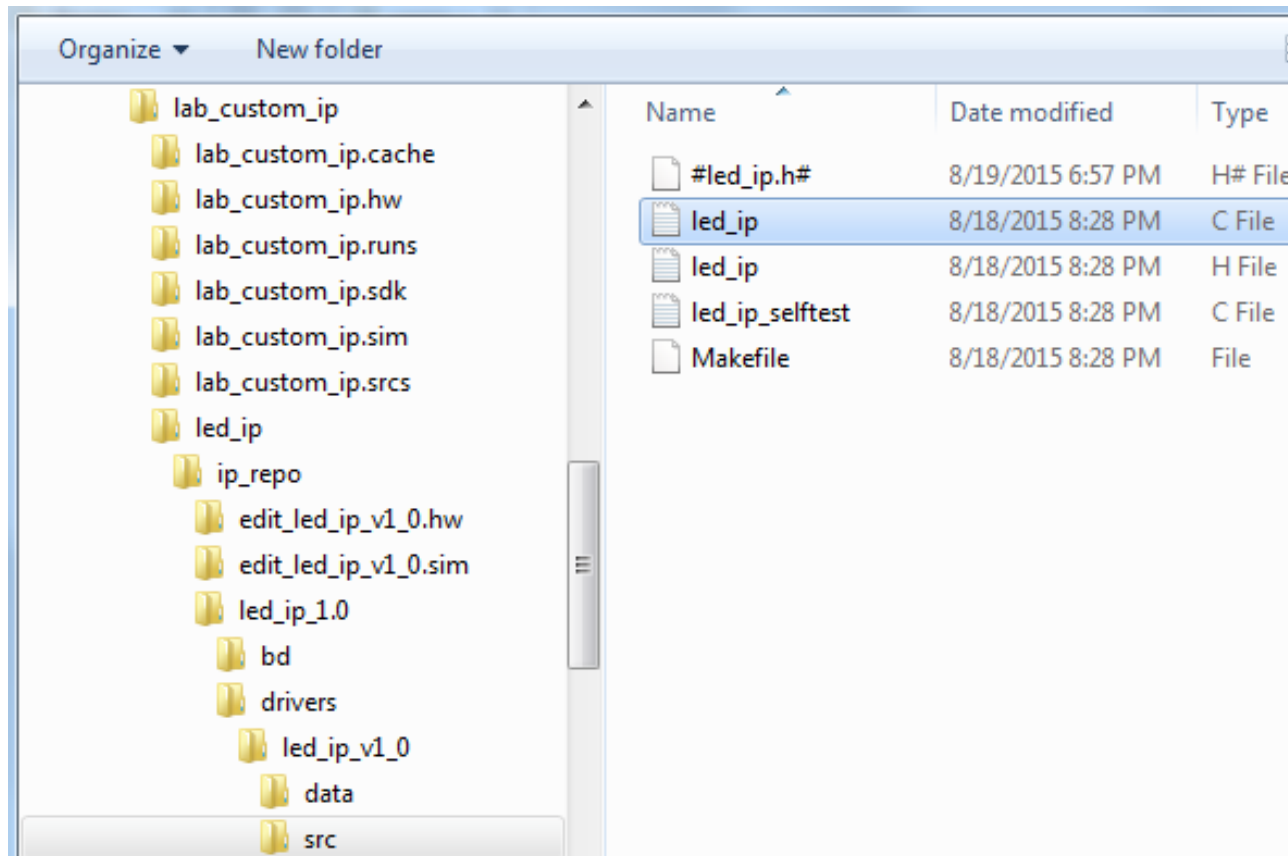
---

- The *driver code* are generated automatically when the IP template is created.
- The *driver* includes higher level functions which can be called from the user application.
- The *driver* will implement the low level functionality used to control your peripheral.

`led_ip\ip_repo\led_ip_1.0\drivers\led_ip_v1_0\src` { `led_ip.c`  
`led_ip.h` { `LED_IP_mWriteReg(...)`  
`LED_IP_mReadReg(...)`

# Custom IP Drivers: \*.c

*led\_ip\ip\_repo\led\_ip\_1.0\drivers\led\_ip\_v1\_0\src\led\_ip.c*



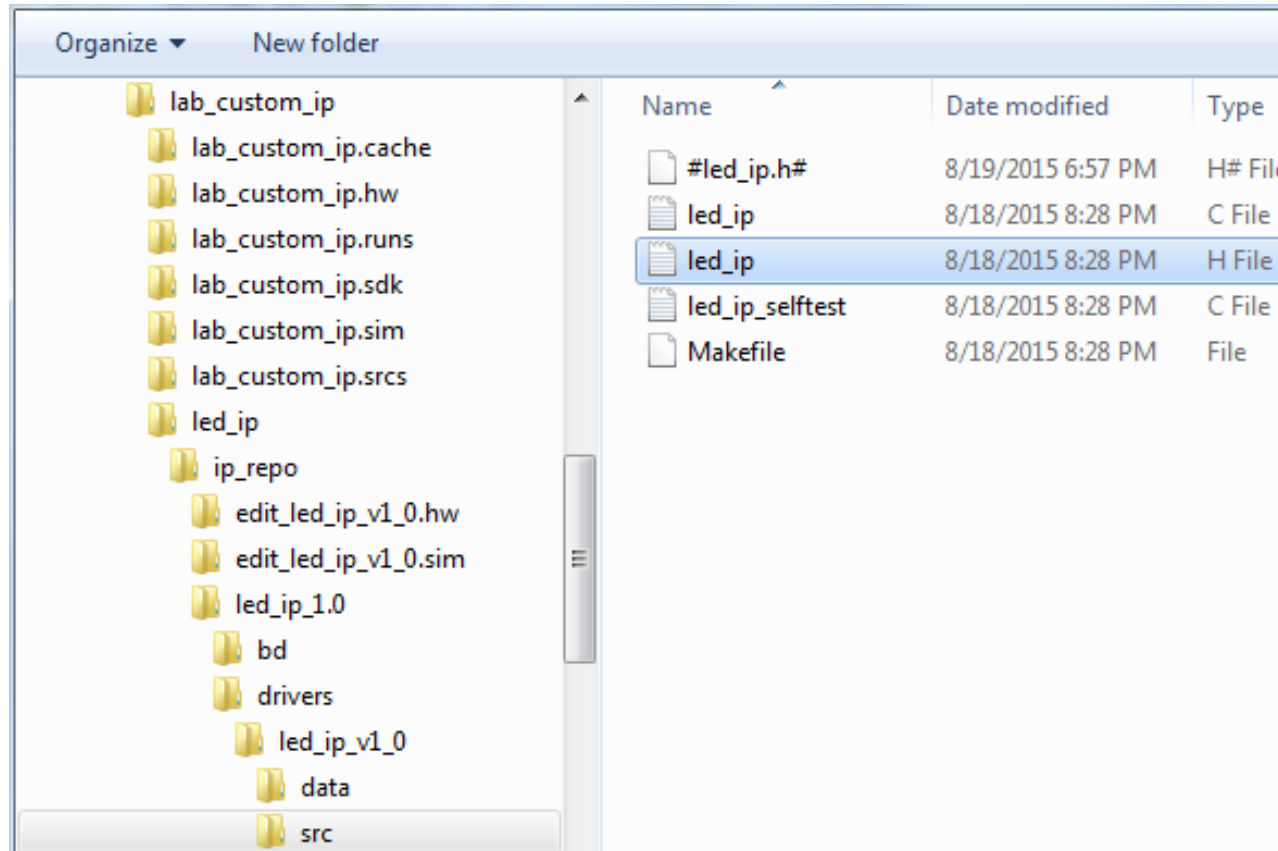
```
led_ip.c
```

```
/****** Include Files *****/
#include "led_ip.h"

/****** Function Definitions *****/
```

# Custom IP Drivers: \*.h

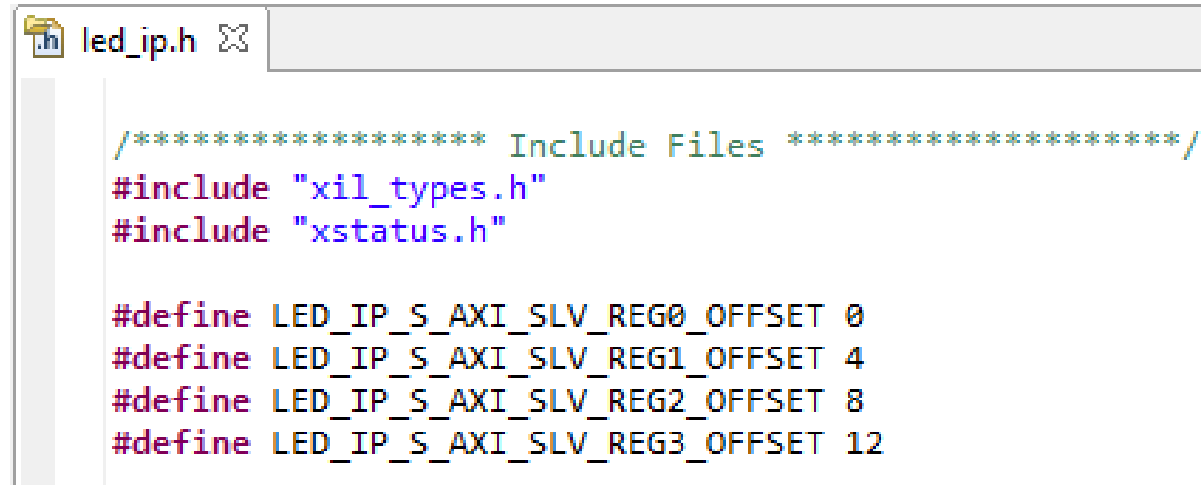
*led\_ip\ip\_repo\led\_ip\_1.0\drivers\led\_ip\_v1\_0\src\led\_ip.h*



# Custom IP Drivers: \*.h (cont' 1)

---

*led\_ip\ip\_repo\led\_ip\_1.0\drivers\led\_ip\_v1\_0\src\led\_ip.h*

A screenshot of a code editor window titled 'led\_ip.h'. The editor shows the following C preprocessor directives:

```
/****** Include Files *****/
#include "xil_types.h"
#include "xstatus.h"

#define LED_IP_S_AXI_SLV_REG0_OFFSET 0
#define LED_IP_S_AXI_SLV_REG1_OFFSET 4
#define LED_IP_S_AXI_SLV_REG2_OFFSET 8
#define LED_IP_S_AXI_SLV_REG3_OFFSET 12
```

# Custom IP Drivers: \*.h (cont' 2)

---

*led\_ip\ip\_repo\led\_ip\_1.0\drivers\led\_ip\_v1\_0\src\led\_ip.h*

```
/**
 *
 * Write a value to a LED_IP register. A 32 bit write is performed.
 * If the component is implemented in a smaller width, only the least
 * significant data is written.
 *
 * @param BaseAddress is the base address of the LED_IPdevice.
 * @param RegOffset is the register offset from the base to write to.
 * @param Data is the data written to the register.
 *
 * @return None.
 *
 * @note
 * C-style signature:
 * void LED_IP_mWriteReg(u32 BaseAddress, unsigned RegOffset, u32 Data)
 *
 */
#define LED_IP_mWriteReg(BaseAddress, RegOffset, Data) \
    Xil_Out32((BaseAddress) + (RegOffset), (u32)(Data))
```

# Custom IP Drivers: \*.h (cont' 3)

---

*led\_ip\ip\_repo\led\_ip\_1.0\drivers\led\_ip\_v1\_0\src\led\_ip.h*

```
/**
 *
 * Read a value from a LED_IP register. A 32 bit read is performed.
 * If the component is implemented in a smaller width, only the least
 * significant data is read from the register. The most significant data
 * will be read as 0.
 *
 * @param   BaseAddress is the base address of the LED_IP device.
 * @param   RegOffset is the register offset from the base to write to.
 *
 * @return  Data is the data from the register.
 *
 * @note
 * C-style signature:
 * u32 LED_IP_mReadReg(u32 BaseAddress, unsigned RegOffset)
 */
#define LED_IP_mReadReg(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (RegOffset))
```

# Custom IP Drivers: \*.h (cont' 4)

---

*led\_ip\ip\_repo\led\_ip\_1.0\drivers\led\_ip\_v1\_0\src\led\_ip.h*

```
/**
 *
 * Run a self-test on the driver/device. Note this may be a destructive test if
 * resets of the device are performed.
 *
 * If the hardware system is not built correctly, this function may never
 * return to the caller.
 *
 * @param  baseaddr_p is the base address of the LED_IP instance to be worked on
 *
 * @return
 *
 * - XST_SUCCESS   if all self-test code passed
 * - XST_FAILURE   if any self-test code failed
 *
 * @note    Caching must be turned off for this function to work.
 * @note    Self test may fail if data memory and device are not on the same bus.
 */
XStatus LED_IP_Reg_SelfTest(void * baseaddr_p);
```

# 'C' Code for Writing to My\_IP

```
#include "xparameters.h"
#include "xgpio.h"
#include "led_ip.h"

//=====
int main (void)
{
    XGpio dip, push;
    int i, psb_check, dip_check;

    xil_printf("-- Start of the Program --\r\n");

    XGpio_Initialize(&dip, XPAR_SWITCHES_DEVICE_ID);
    XGpio_SetDataDirection(&dip, 1, 0xffffffff);

    XGpio_Initialize(&push, XPAR_BUTTONS_DEVICE_ID);
    XGpio_SetDataDirection(&push, 1, 0xffffffff);

    while (1)
    {
        psb_check = XGpio_DiscreteRead(&push, 1);
        xil_printf("Push Buttons Status %x\r\n", psb_check);
        dip_check = XGpio_DiscreteRead(&dip, 1);
        xil_printf("DIP Switch Status %x\r\n", dip_check);

        // output dip switches value on LED_ip device
        LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, dip_check);

        for (i=0; i<9999999; i++);
    }
}
```



# IP Drivers – *Xil\_Out32/Xil\_In32*

---

```
#define LED_IP_mWriteReg(BaseAddress, RegOffset, Data) Xil_Out32((BaseAddress) + (RegOffset), (Xuint32)(Data))
```

```
#define LED_IP_mReadReg(BaseAddress, RegOffset) Xil_In32((BaseAddress) + (RegOffset))
```

- For this driver, you can see the macros are aliases to the lower level functions **Xil\_Out32( )** and **Xil\_In32( )**
- The macros in this file make up the higher level API of the led\_ip driver.
- If you are writing your own driver for your own IP, you will need to use low level functions like these to read and write from your IP as required. The low level hardware access functions are wrapped in your driver making it easier to use your IP in an Application project.

# IP Drivers – *Xil\_In32 (xil\_io.h/xil\_io.c)*

---

```

/*****
**
* Performs an input operation for a 32-bit memory location by reading from the
* specified address and returning the Value read from that address.
*
* @param    Addr contains the address to perform the input operation at.
*
* @return    The Value read from the specified input address.
*
* @note      None.
*
*****/
u32 Xil_In32(INTPTR Addr)
{
    return *(volatile u32 *) Addr;
}

```

# IP Drivers – *Xil\_Out32 (xil\_io.h/xil\_io.c)*

---

```

/*****
/**
 * Performs an output operation for a 32-bit memory location by writing the
 * specified Value to the the specified address.
 *
 * @param   Addr contains the address to perform the output operation at.
 * @param   Value contains the Value to be output at the specified address.
 *
 * @return  None.
 *
 * @note    None.
 *****/
void Xil_Out32(INTPTR Addr, u32 Value)
{
    u32 *LocalAddr = (u32 *)Addr;

    *LocalAddr = Value;
}

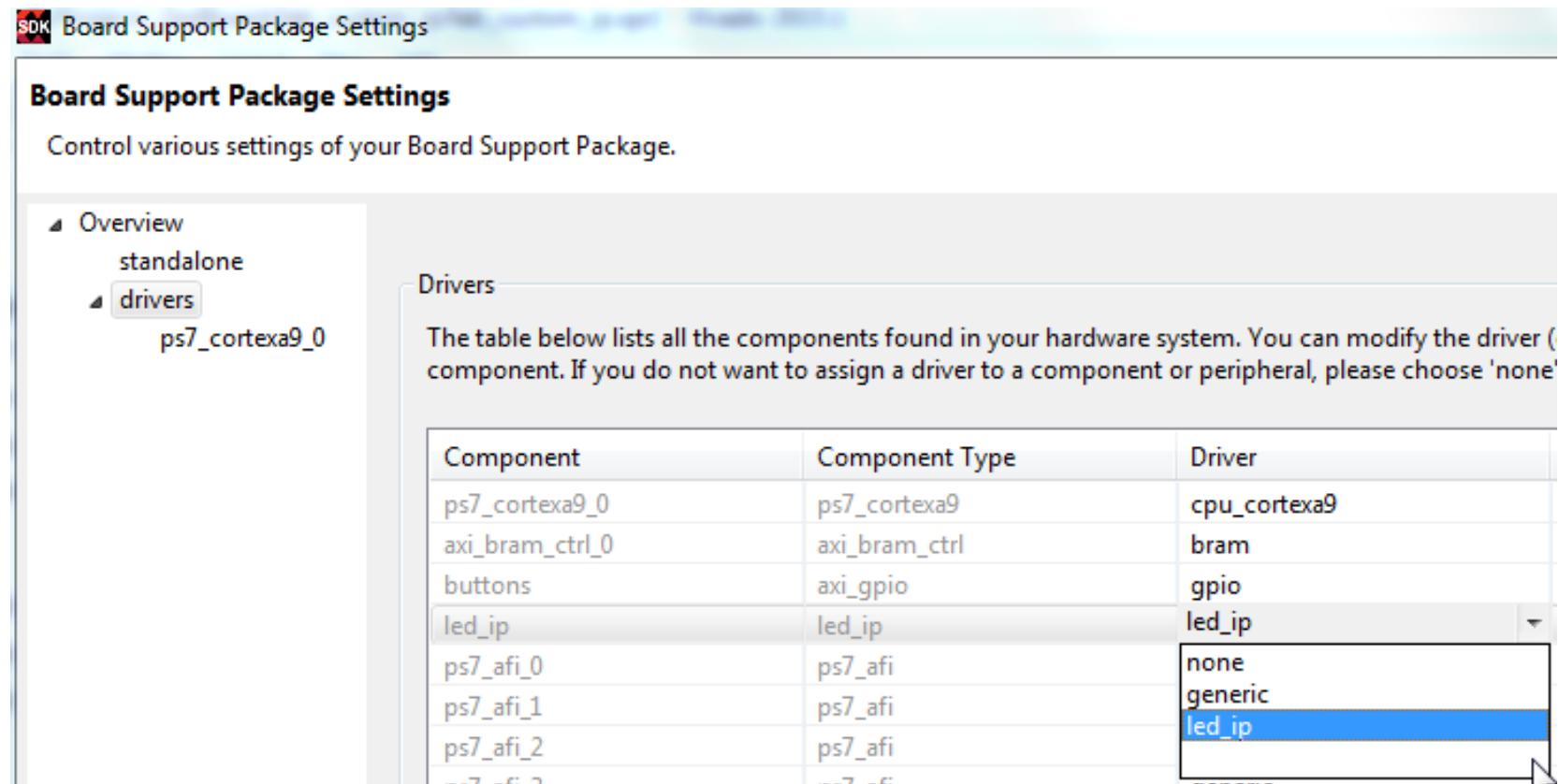
```

# IP Drivers – SDK ‘Activation’

---

- Select **<project\_name>\_bsp** in the project view pane. Right-click
- Select **Board Support Package Settings**
- Select **Drivers** on the **Overview** pane
- If the **led\_ip** driver has not already been selected, select Generic under the Driver Column for **led\_ip** to access the dropdown menu. From the dropdown menu, select **led\_ip**, and click OK>

# IP Drivers – SDK 'Activation' (cont')



**Board Support Package Settings**

Control various settings of your Board Support Package.

**Overview**

- standalone
- drivers**
- ps7\_cortexa9\_0

**Drivers**

The table below lists all the components found in your hardware system. You can modify the driver (component). If you do not want to assign a driver to a component or peripheral, please choose 'none'

Component	Component Type	Driver
ps7_cortexa9_0	ps7_cortexa9	cpu_cortexa9
axi_bram_ctrl_0	axi_bram_ctrl	bram
buttons	axi_gpio	gpio
led_ip	led_ip	led_ip
ps7_afi_0	ps7_afi	none
ps7_afi_1	ps7_afi	generic
ps7_afi_2	ps7_afi	led_ip
ps7_afi_3	ps7_afi	

---

# Read and Write From/To Memory

---

# Note On Reading from / Writing to Memory

---

Generally speaking processors work on byte (8bit) address boundaries.

If we wish to write byte-wide data values into the first four consecutive locations in a region of memory starting at "*DDR\_BASEADDR*", we must write the first to *DDR\_BASEADDR + 0*, the second to *DDR\_BASEADDR + 1*, the third to *DDR\_BASEADDR + 2*, and the last to *DDR\_BASEADDR + 3*.

However, if we wish to write four half-word wide (16 bit) data values to four memory addresses starting at the same location, we must write the first to *DDR\_BASEADDR + 0*, the second to *DDR\_BASEADDR + 2*, the third to *DDR\_BASEADDR + 4*, and the last to *DDR\_BASEADDR + 6*.

***When writing word wide (32 bit) data values, we must do so on 4 byte boundaries; 0x0, 0x4, 0x8, and 0xC.***

# Reading from / Writing to Memory: *xil\_io.h*

---

## ***Writing Functions***

***Xil\_Out8***(memory\_address, 8\_bit\_value);

***Xil\_Out16***(memory\_address, 16\_bit\_value);

***Xil\_Out32***(memory\_address, 32\_bit\_value);

## ***Reading Functions***

8\_bit\_value = ***Xil\_In8***(memory\_address);

16\_bit\_value = ***Xil\_In16***(memory\_address);

32\_bit\_value = ***Xil\_In32***(memory\_address);



# Reading from / Writing to Memory: *xil\_io.h*

---

```
int main(void)
{
    int result1; // integers are 32 bits wide!
    int result2; // integers are 32 bits wide!

    Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 0, 0x12);
    Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 1, 0x34);
    Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 2, 0x56);
    Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 3, 0x78);
    result1 = Xil_In32(XPAR_PS7_RAM_0_S_AXI_BASEADDR);
    Xil_Out16(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 4, 0x9876);
    Xil_Out16(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 6, 0x5432);
    result2 = Xil_In32(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 4);
    return(0);
}
```