The Abdus Salam International Centre for Theoretical Physics

IAEA unesco

Secretaría Nacional de Ciencia y Tecnología

IUPAP INTERNATIONAL UNION OF PURE AND APPLIED PHYSICS

USAC TRICENTENARIA Universidad de San Carlos de Guatemala

1st Mesoamerican Workshop on Reconfigurable X-ray Scientific Instrumentation for Cultural Heritage

‘C’ for Embedded Systems

Cristian Sisterna

Senior Associate – ICTP MLAB

Universidad Nacional de San Juan

What is ‘Embedded C’ ?

Embedded C is a **set of language extensions for the C programming language** designed specifically for programming **embedded systems** — small computing devices that control hardware in real-time.

It’s **not a separate language**, but rather **C tailored for embedded applications** with additional features to support direct interaction with hardware.

Embedded C is essentially **C adapted to run "closer to the I/Os"** — lean, efficient, and tightly integrated with hardware

Differences Between ‘C’ and ‘*Embedded C*’

Feature	Regular C	Embedded C
🔧 **Target System**	General-purpose computers (PCs, servers)	Microcontrollers, embedded systems
🖥️ **Operating System**	Often relies on OS (e.g., Linux, Windows)	Often no OS or a Real-Time OS (RTOS)
🔗 **Libraries**	Standard C libraries (stdio.h, etc.)	Limited or custom libraries; often no I/O streams
📡 **Hardware Access**	Abstracted from hardware	Direct register and port manipulation
🧠 **Memory Usage**	Abundant (RAM, Disk)	Very limited memory (few KB to MB)
🕒 **Timing**	Not deterministic	Precise, deterministic timing often needed
💻 **I/O Handling**	Through OS APIs or files	Direct I/O via registers (e.g., `PORTA
📦 **Compilation**	Compiles to run on the host system	Cross-compiled for a specific microcontroller
⚙ **Toolchains**	GCC, Clang	Keil, MPLAB, IAR, AVR-GCC, etc.
🛠️ **Typical Use Cases**	Software apps, games, compilers	Device drivers, firmware, real-time control

Difference Between ‘C’ and ‘*Embedded C*’

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.

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

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

Advantages of Using *Embedded C*

Feature	Description
🧠 **Efficiency**	Designed for low-level access and minimal resource usage.
🔌 **Hardware Access**	Supports direct access to hardware registers and I/O ports.
🕒 **Real-time Capable**	Used in systems that require deterministic timing.
🛠️ **Portability**	Code can often be reused across microcontrollers with minor changes. Unlike assembly.
⚙ **Extensions**	Compiler-specific features like __interrupt, __bit, __sfr etc. allow low-level control.
💻 **I/O access**	It supports access to I/O and provides ease of management of large embedded projects

Reviewing Embedded ‘C’ Basic Concepts

‘C’ Basic Data Types

Data Type	Description	Size (Typical)	Format Specifier
int	Integer (whole numbers)	4 bytes	%d
char	Character	1 byte	%c
float	Floating point (single precision)	4 bytes	%f
double	Floating point (double precision)	8 bytes	%lf
void	No value (used for functions that return nothing)	N/A	N/A

‘C’ Derived Data Types & Modifiers

These are **derived from basic types**:

- ✓ **Arrays** (e.g., int arr[10];)
- ✓ **Pointers** (e.g., int *p;)
- ✓ **Structures** (e.g., struct Person { ... };)
- ✓ **Unions** (e.g., union Data { ... };)
- ✓ **Functions** (e.g., int func(int x);)

You can **modify** basic types with the following **type qualifiers**:

- ➢ short
- ➢ long
- ➢ signed
- ➢ unsigned

Modified Type	Typical Size	Notes
short int	2 bytes	Smaller range of integers
long int	4 or 8 bytes	Larger range
unsigned int	4 bytes	Only non-negative values
long double	12 or 16 bytes	Higher precision float

Xilinx-AMD ‘C’ Basic Data Types

xbasic_types.h

This file contains basic types for Xilinx software IP.

```c
/** @name Legacy types
 * Deprecated legacy types.
 * @{
 */
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) */
```

Xilinx-AMD ‘C’ Basic Data Types

xil_types.h

The ***xil_types.h*** file contains basic types for Xilinx software IP. These data types are applicable for all processors supported by Xilinx.

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

```c
typedef char char8;
typedef int8_t s8;
typedef int16_t s16;
typedef int32_t s32;
typedef int64_t s64;
typedef uint64_t u64;
typedef int sint32;
```

Use of `#include` directive

#include is a **directive** that is used to *include* the contents of a file (usually a header file, .h file) into your source code.

The syntax for the **#include** directive can use either double quotes (" ") or angle brackets (< >), and there are important differences between the two:

#include <filename> (Angle Brackets)
Search Path: When you use angle brackets, the preprocessor searches for the specified file only in the standard system directories (e.g., /usr/include on Unix/Linux systems). *It does not look in the current directory.*
Usage: This is generally used for including standard library headers or system headers that are part of the C standard library.

```
#include <stdio.h>
```

#include "filename" (Double Quotes)
Search Path: When you use double quotes, the preprocessor first searches for the specified file in the same directory as the source file that contains the #include directive. If the file is not found there, it then searches the standard system directories.
Usage: This is typically used for including user-defined header files or files that are part of your project.

```
#include “my_header”;
```

`#ifndef` directive

`#ifndef` is a **directive** that stands for **"if not defined"**.
It's used to prevent **multiple inclusion** of the same header file, which can cause compilation errors.

```
#ifndef IDENTIFIER
#define IDENTIFIER



#endif
```

- ✓ **#ifndef** checks if the identifier (macro) has **not been defined yet**.
- ✓ If it hasn't, the code inside the block is included.
- ✓ **#define** then marks it as defined, so the next time the file is included, the code is skipped.

Using **#ifndef** + **#define** is called a *header guard*, and it's a best practice in C/C++ programming.

Local vs Global Variables

In C programming, **variables** can be **local** or **global** depending on **where** they are declared and **how** they are accessed.

Local Variables

Local variables are **declared inside a function**, block, or compound statement and are **accessible only within that scope**.

- ✓ Accessible only by the function within which they are declared
- ✓ Created when the function is called.
- ✓ Destroyed when the function exits.
- ✓ Not accessible outside their scope

Global Variables

Global variables are declared **outside of all functions**, usually at the top of the program file. They are **accessible from any function** in the program.

- ✓ **Declared outside** any function.
- ✓ **Exist for the lifetime** of the program.
- ✓ Can be **accessed or modified by any function**

Global and Local Variables Declarations

```
int flag  = 0;
char note = 'a';

main ()
{
     ...
     flag = 1;
     function1( );
     ...
     flag = 2;
     …
}

int function1()
{
int alarm = 128;
     ...
     alarm =+1;
     flag  = 3;
     ...
}
```

‘C’ Modifiers

In **C language**, **modifiers** are keywords that **modify the meaning or behavior of *variables*, *functions*, and *data types***.

They can affect **storage**, **visibility**, **lifetime**, **type size**, and **optimization behavior**.

'C' Modifiers

- Storage-Class Modifiers
- Type Modifiers

‘C’ Modifiers - Storage-Class Modifiers

These control the **lifetime**, **scope**, and **linkage** of variables or functions.

Modifier	Purpose	Notes
auto	Default for local variables (rarely used explicitly)	
register	Hints to store variable in a CPU register (deprecated in modern compilers)	
static	Keeps variable's value across function calls / restricts visibility to file	Retains value, restricts linkage.
extern	Declares a variable/function defined in another file	Share variables/functions between files.
volatile	Prevents compiler optimization; ensures variable is read from memory every time	Useful for variables that changes outside normal control (e.g. hardware).

Use of the '*static*' modifier with variables

2 ❖The '**static**' 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

1 ❖ The '**static**' 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)

2
```
static int flag  = 0;
static char note = 'a';

main  ()
{
      ...
      flag = 1;
      function1( );
      ...
      flag = 2;
      ...
}

int function1()
{
static int alarm = 128;
      …
      alarm =+1;
      flag  = 3;
      ..
}
```
1

Use of the ‘*static*’ modifier with functions

- The ‘**static**’ modifier in a function declaration causes that the functions is only callable within the file where is declared.

```
static void helper() {

    // only callable within this file

}
```

‘*volatile*’ Variable

Tells the compiler **not to optimize** the variable because its value can change unexpectedly (e.g. interrupts, hardware registers).

Ensure each access actually read or write the memory location.

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***.

‘*volatile*’ Variable Example

```c
volatile int sensorFlag;

void checkSensor() {
    while (sensorFlag == 0) {
        // wait for flag to change (e.g., set by ISR)
    }
    // sensorFlag has been set – handle it
}
```

*Without **volatile** the compiler might optimize the loop away because it assumes sensorflag variable **never changes**.*

Use of the ‘*static*’ and ‘*volatile*’ modifiers

Why Combine static and volatile?

- **volatile** tells the compiler
  - “This variable can change at any time (outside normal program flow, like via an interrupt), so don’t optimize accesses to it.”
- **static** ensures the variable
  - “Persists between function calls and is only visible within this file (or function).”

Use of the '*static*' and '*volatile*' modifiers

Example: You have a **button interrupt** that sets a **flag**. The 'C' **main loop** waits for this flag to change to take action.

static
- File-level scope
- Keep buttonpressed local to the file

volatile
- Prevents optimization
- Ensures compiler does not cache the variable value, reads from memory every time

```c
// Static + volatile flag shared between ISR and main loop
static volatile bool buttonPressed = false;

// ISR: gets called when button is pressed
void __interrupt() button_isr(void) {
    buttonPressed = true;  // Set the flag (from interrupt context)
}

// Main loop
int main(void) {
    while (1) {
        if (buttonPressed) {
            // Clear flag and handle the button press
            buttonPressed = false;
            // Do something, e.g., toggle LED
        }
    }
    return 0;
}
```

‘C’ Modifiers - Type Modifiers

These modify the **size** or **sign** of **data types**.

Modifier	Purpose
signed	Default for int/char: can hold negative and positive values
unsigned	Only positive values (doubles the upper limit)
short	Smaller-sized integer (usually 16 bits)
long	Larger-sized integer (usually 32 or 64 bits)
long long	Even larger integer (usually 64 bits)

Functions Data Types

Function data types refer to the types of values that functions can return and the types of parameters they can accept.

Return Type: Every function in C has a return type that specifies the type of value the function will return.

Common **return** types include:
int: Returns an integer value.
float: Returns a floating-point value.
double: Returns a double-precision floating-point value.
char: Returns a character.
void: Indicates that the function does not return a value.

Parameter Types: Functions can accept parameters of various data types. The types of parameters must be specified in the function definition. You can have multiple parameters of different types.

```
void printSum(int a, float b) {
    printf("Sum: %f\n", a + b);
}
```

Functions Data Types

Function Pointers: In C, you can also define pointers to functions, which allows you to store the address of a function and call it later.

The type of a function pointer is defined by the return type and the parameter types.

```c
#include <stdio.h>
// Function that takes two integers and returns their sum
int add(int a, int b) {
    return a + b;
}
// Function that takes two integers and returns their product
int multiply(int a, int b) {
    return a * b;
}
int main() {
    // Declare a function pointer that takes two integers and returns an integer
    int (*operation)(int, int);
    // Assign the address of the 'add' function to the function pointer
    operation = &add;
    printf("Addition: %d\n", operation(5, 3)); // Calls add(5, 3)
    // Assign the address of the 'multiply' function to the function pointer
    operation = &multiply;
    printf("Multiplication: %d\n", operation(5, 3)); // Calls multiply(5, 3)
    return 0;
}
```

Structures

In C programming language, a **structure** (struct) is a ***user-defined data type*** that allows you to group different types of variables under a single name.

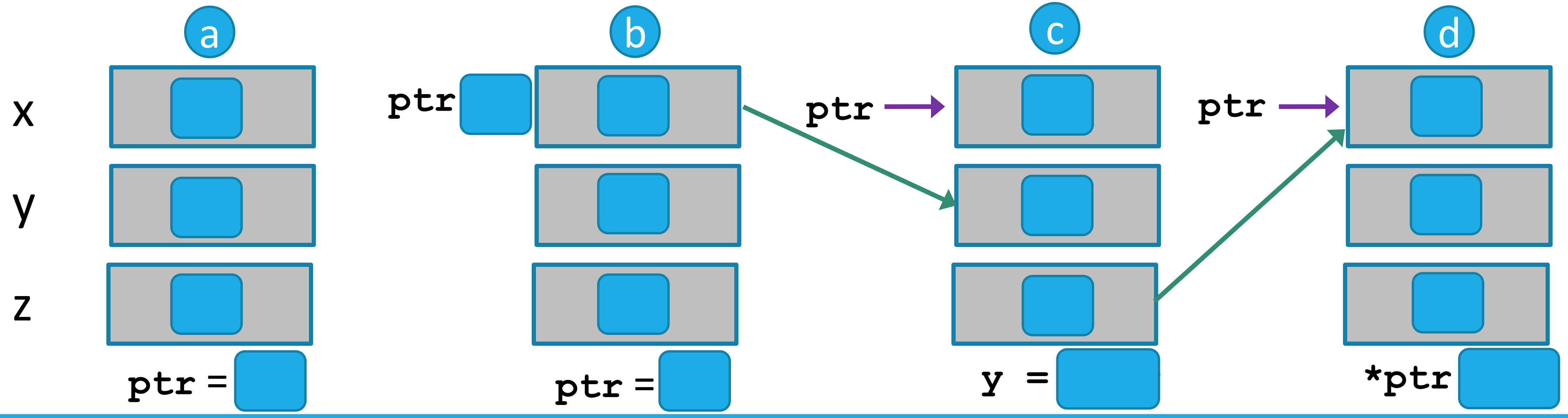
```c
#include "xparameters.h"
#include "xgpio.h"
#include "xgpiops.h"

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

//====================================

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

```c
/**
 * 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;
```

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;
```

```
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.

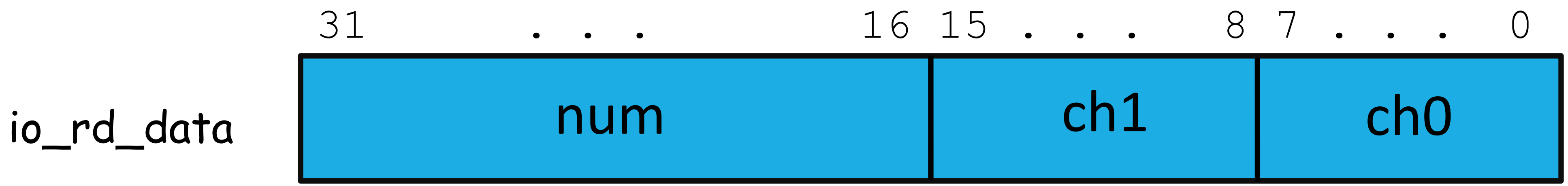
Bit Shift Example

```
void led_(XGpio *pLED_GPIO, int nNumberOfTimes)
{
        int i=0;        int j=0;
        u8 uchLedStatus=0;
        //
        for(i=0;i<nNumberOfTimes;i++)
        {
                for(j=0;j<8;j++)   //
                {
                        uchLedStatus = 1 << j;
                        XGpio_DiscreteWrite(pLED_GPIO, 1, uchLedStatus);
                        delay(ABOUT_ONE_SECOND / 15);
                }
                for(j=0;j<8;j++)   //
                {
                        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(...);//my_io_read read a data
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

	31 . . . 16	15 . . . 8	7 . . . 0
io_wr_data	num	ch1	ch0

```
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]
```

Another Way ....

```
wr_data = (((u32)(num))<<16)|(((u32)ch1)<<8)|(u32)ch2;
```

Basic Embedded ‘C’ Program Template

Embedded System Application

In **embedded systems**, applications are typically designed as a collection of **tasks** or **functional blocks**, each responsible for a specific operation. These tasks can be implemented using:

Software Routines

- ✓ Executed by a general-purpose processor (e.g., ARM Cortex).
- ✓ Written in C/C++ or assembly.
- ✓ Good for tasks that are:
  - ✓ Control-intensive
  - ✓ Low-throughput
  - ✓ Complex to parallelize

Hardware Accelerators

- ✓ Implemented on FPGAs, ASICs, or dedicated coprocessors.
- ✓ Designed using RTL (VHDL/Verilog) or HLS (C/C++ → Hardware).
- ✓ Best for tasks that are:
  - ✓ Compute-intensive
  - ✓ Highly parallel
  - ✓ Time-critical

Example Embedded System Application

Task	Implementation
Frame capture	Software
Color space conversion	Hardware (HLS)
Edge detection	Hardware (RTL/HLS)
Display output	Software

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();
           }
  }
```

I/O Simple 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

I/O Simple 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);
         }
   }
```

I/O Simple Example - Reading

```
/**********************************************************************
* function: read_sw ()
* purpose: get flashing period from 10 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) & 0x000003ff;  //read flashing period
                                                // from switch
}
```

I/O Simple 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 : none
* 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
}
```

I/O Example – Read / Write

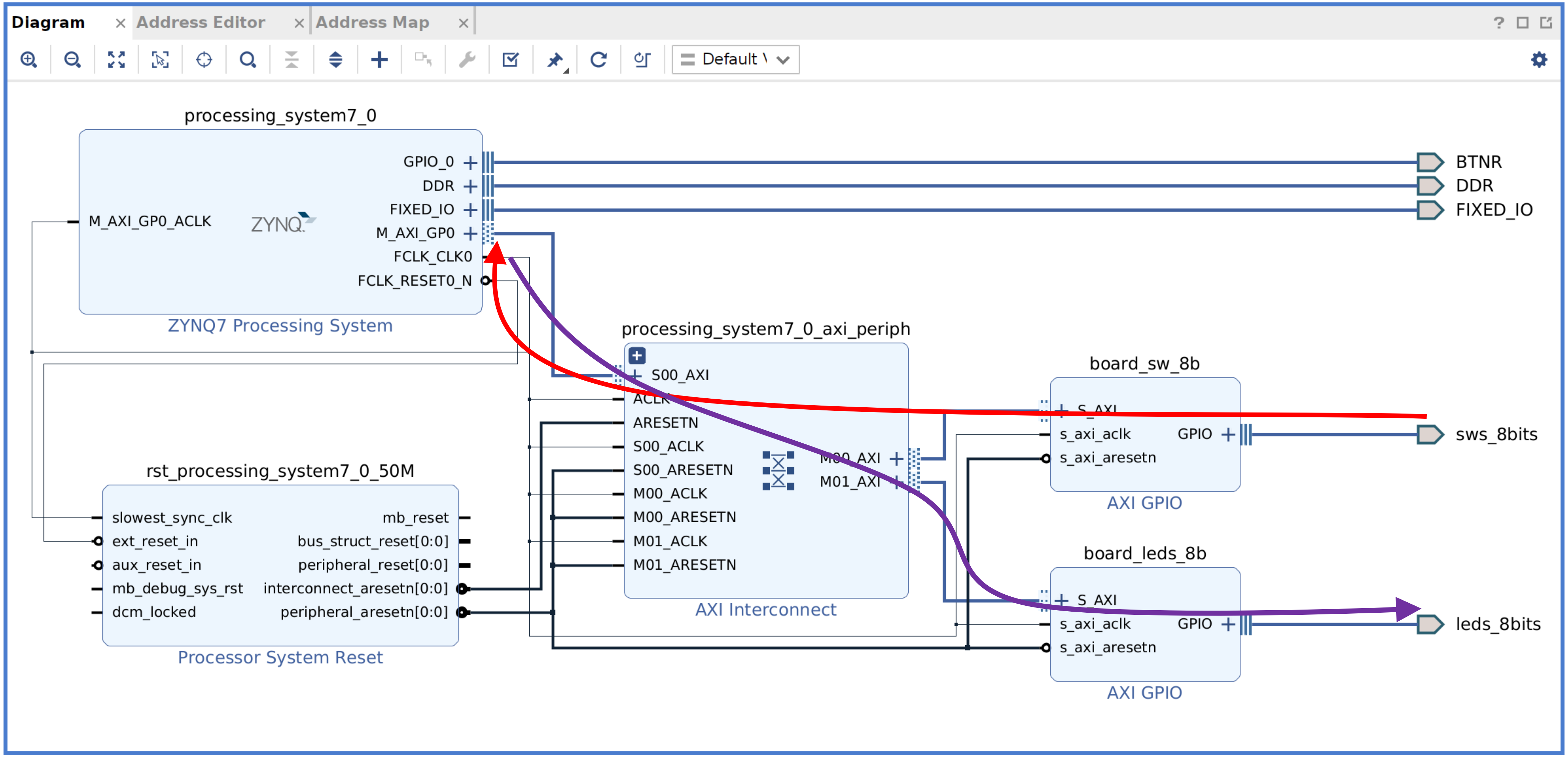
```c
int main()
  {
  int period;

  while(1){
    read_sw(SWITCH_S1_BASE, &period);
    led_flash(LED_L1_BASE, period);
          }
  return 0;
  }
```

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

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

Zynq PSoC: Read/Write From/To GPIO Inputs and Outputs

Example of Wr/Rd to/from GPIO



Steps for Reading from a GPIO

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

Steps for Reading from a GPIO

1. Create a GPIO instance
2. Initialize the GPIO
3. Set data direction (optional)
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;
```

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 (already declared).

DeviceID: is the unique **ID** of the device controlled by this **XGpio** component (declared in the ***xparameters.h*** file)

@return

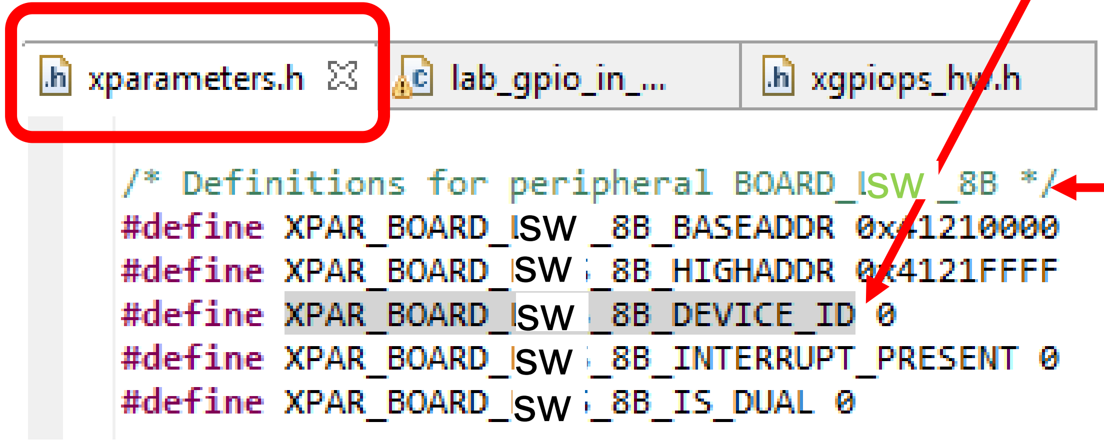
- XST_SUCCESS if the initialization was successfull.
- XST_DEVICE_NOT_FOUND if the device configuration data was not

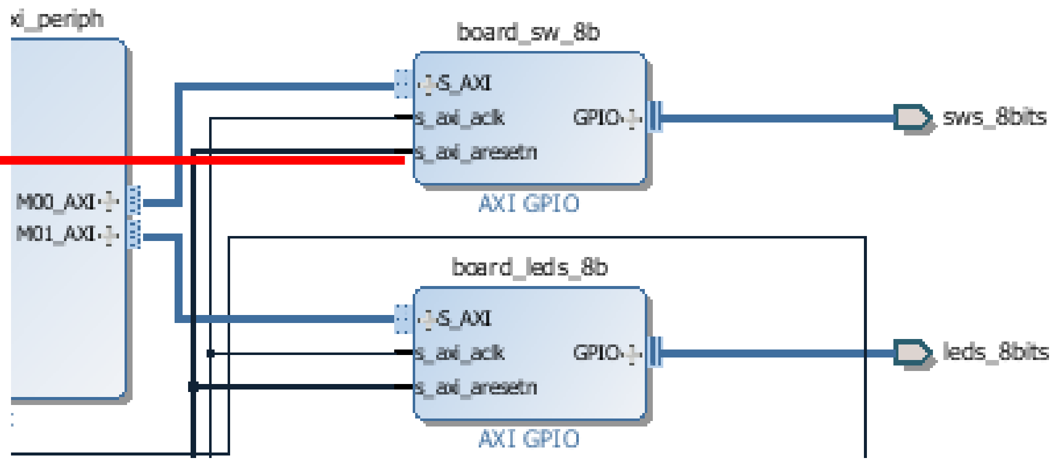
xstatus.h

Steps for Reading from a GPIO – Step 2

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

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

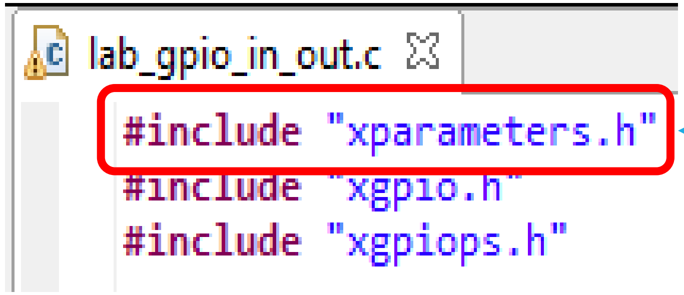




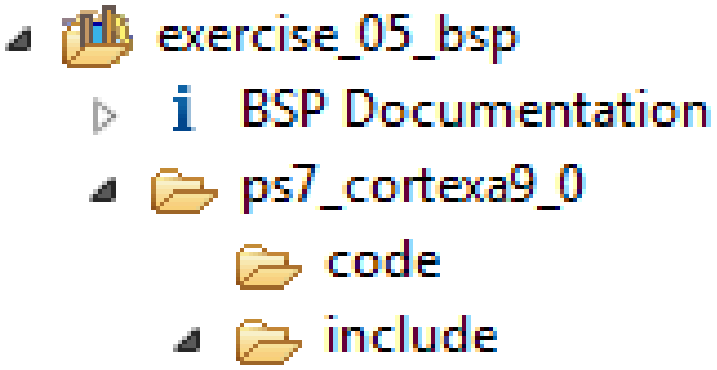
xparameters.h

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

This file is generated from the hardware platform created in Vivado

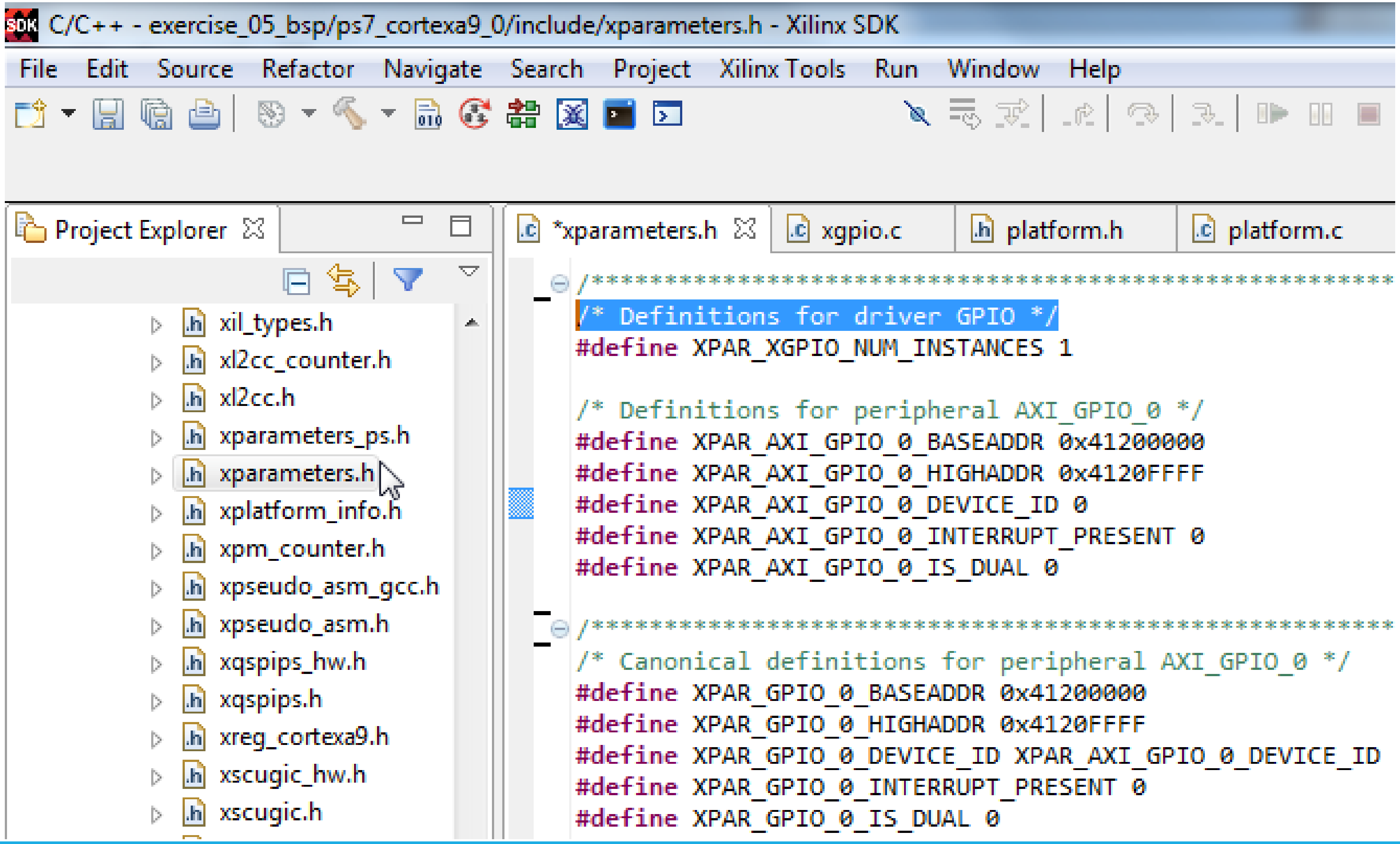


Ctrl + Mouse Over



xparameters.h file can be found underneath the include folder in the ps7_cortexa9_0 folder of the BSP main folder

xparameters.h



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 working with.

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

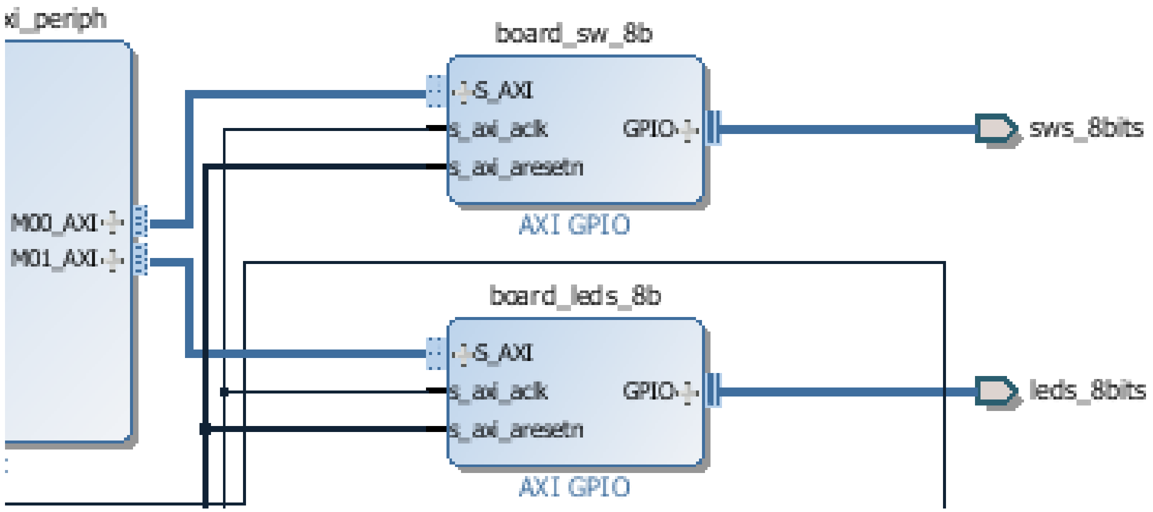
DirectionMask: is a bitmask specifying which bits are inputs and which are outputs. Bits set to **‘0’ are output**, bits set to **‘1’ are inputs**.

Return: none

Steps for Reading from a GPIO – Step 3

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

```
// AXI GPIO switches: bits direction configuration
XGpio_SetDataDirection(&board_sw_8b, 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 working with.

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

Return: read data

Steps for Reading from a GPIO – Step 4

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

```
// AXI GPIO: read data from the switches
sw_check = XGpio_DiscreteRead(&board_sw_8b, 1);
```

Steps for Writing to GPIO

1. Create a GPIO instance
2. Initialize the GPIO
3. Set the data direction (optional)
4. 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;
```

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.

DeviceID: is the unique id of the device controlled by this XGpio component

@return

- XST_SUCCESS if the initialization was successfull.
- XST_DEVICE_NOT_FOUND if the device configuration data was not

xstatus.h

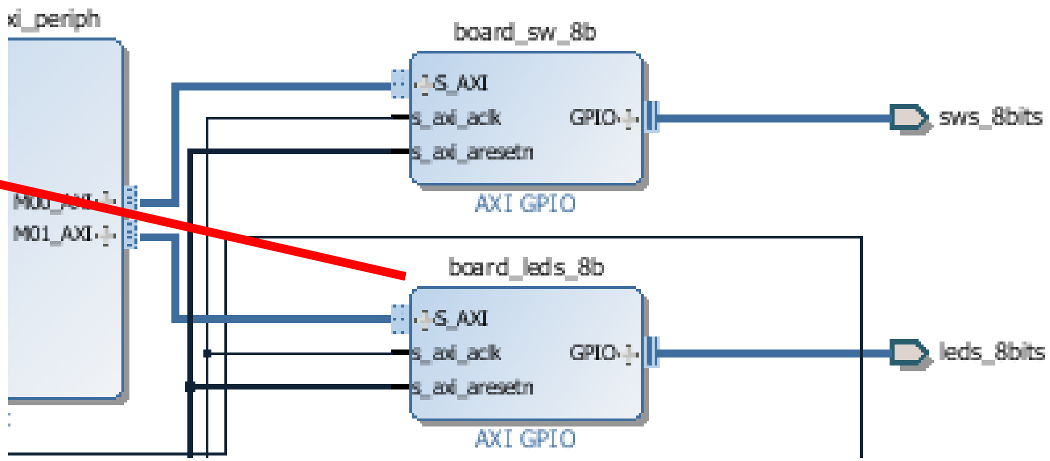
Steps for Writing to a GPIO – Step 2

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

```
// AXI GPIO leds initialization
XGpio_Initialize (&board_leds_8b, XPAR_BOARD_LEDS_8B_DEVICE_ID);
```

xparameters.h | lab_gpio_in_... | xgpiops_hw.h

```
/* Definitions for peripheral BOARD_LEDS_8B */
#define XPAR_BOARD_LEDS_8B_BASEADDR 0x41210000
#define XPAR_BOARD_LEDS_8B_HIGHADDR 0x4121FFFF
#define XPAR_BOARD_LEDS_8B_DEVICE_ID 0
#define XPAR_BOARD_LEDS_8B_INTERRUPT_PRESENT 0
#define XPAR_BOARD_LEDS_8B_IS_DUAL 0
```



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 o 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

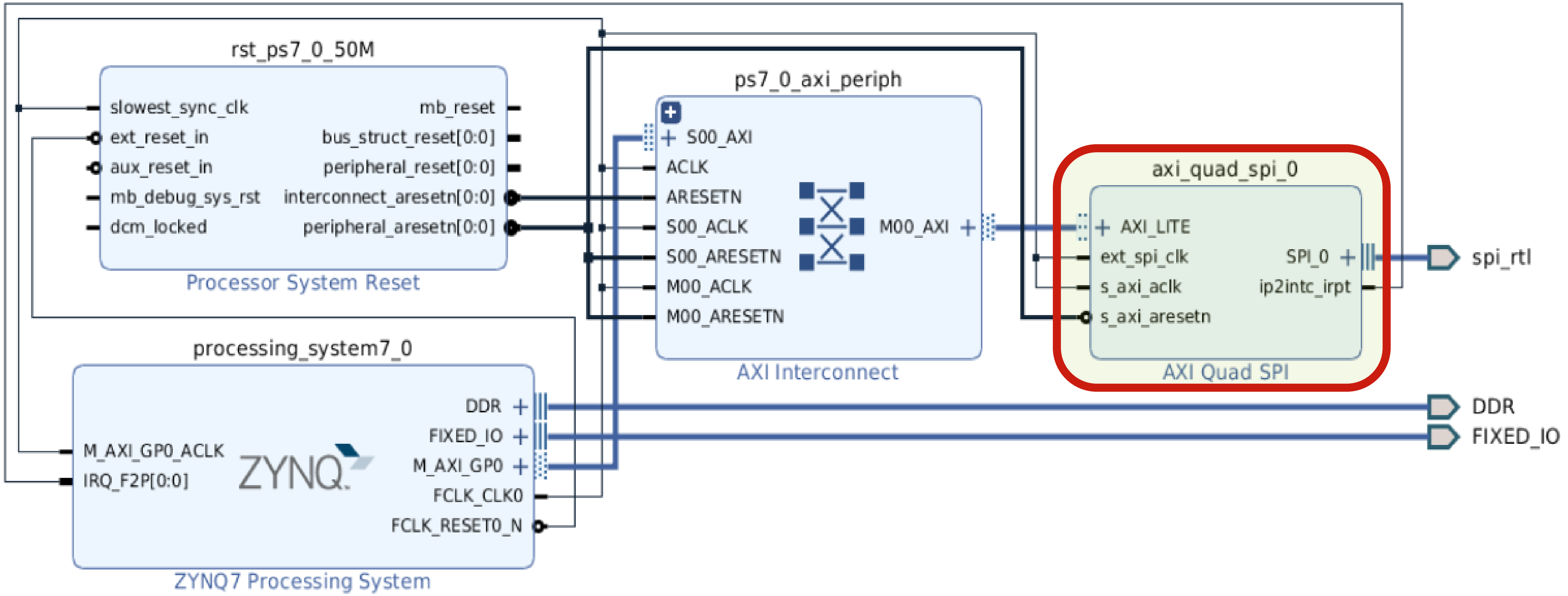
```
void XGpio_DiscreteWrite (XGpio *InstancePtr, unsigned Channel, u32 Data);
```

```
// AXI GPIO: write data (sw_check) to the LEDs
XGpio_DiscreteWrite(& board_leds_8b,1, sw_check);
```

Complete GPIO Rd/Wr Example

‘C’ Drivers for IP Cores

SPI IP Core - Example



SPI IP Core - Example

```c
#include "xparameters.h"
#include "xscugic.h"
#include "xil_exception.h"
#include <stdio.h>
#include "xspi.h"        /* SPI device driver */
```

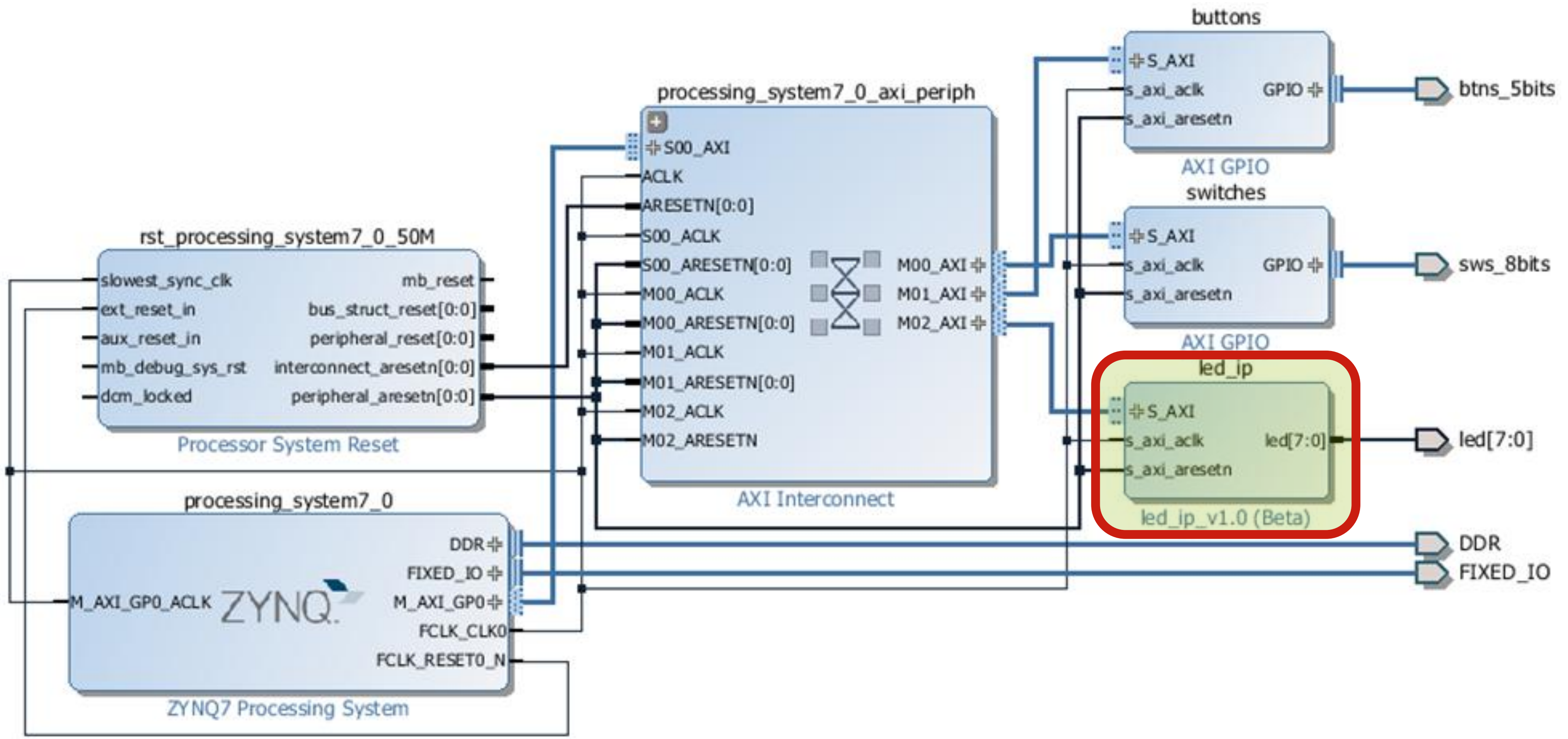
```c
// ------------------- SPI related functions --------------------- //
// Initialize the SPI driver
SPI_ConfigPtr = XSpi_LookupConfig(XPAR_AXI_QUAD_SPI_0_DEVICE_ID);
if (SPI_ConfigPtr == NULL) return XST_DEVICE_NOT_FOUND;

Status = XSpi_CfgInitialize(&SpiInstance, SPI_ConfigPtr, SPI_ConfigPtr->BaseAddress);
if (Status != XST_SUCCESS) return XST_FAILURE;

// Reset the SPI peripheral
XSpi_Reset(&SpiInstance);
```

SPI IP Core - Example

```
/******************************************************************************/
/**
*
* Initializes a specific XSpi instance such that the driver is ready to use.
*
* The state of the device after initialization is:
*   - Device is disabled
*   - Slave mode
*   - Active high clock polarity
*   - Clock phase 0
*
* @param    InstancePtr is a pointer to the XSpi instance to be worked on.
* @param    Config is a reference to a structure containing information
*       about a specific SPI device. This function initializes an
*       InstancePtr object for a specific device specified by the
*       contents of Config. This function can initialize multiple
*       instance objects with the use of multiple calls giving
        different Config information on each call.
* @param    EffectiveAddr is the device base address in the virtual memory
*       address space. The caller is responsible for keeping the
*       address mapping from EffectiveAddr to the device physical base
*       address unchanged once this function is invoked. Unexpected
*       errors may occur if the address mapping changes after this
*       function is called. If address translation is not used, use
*       Config->BaseAddress for this parameters, passing the physical
*       address instead.
*
* @return
*       - XST_SUCCESS if successful.
*       - XST_DEVICE_IS_STARTED if the device is started. It must be
*         stopped to re-initialize.
*
* @note     None.
*
******************************************************************************/
int XSpi_CfgInitialize(XSpi *InstancePtr, XSpi_Config *Config,
            UINTPTR EffectiveAddr)
```

‘C’ Drivers for Custom IP

Custom IP



buttons
S_AXI
s_axi_aclk
GPIO
s_axi_aresetn
AXI GPIO
btns_5bits
switches
S_AXI
s_axi_aclk
GPIO
s_axi_aresetn
AXI GPIO
sws_8bits
led_ip
S_AXI
s_axi_aclk
led[7:0]
s_axi_aresetn
led_ip_v1.0 (Beta)
led[7:0]
processing_system7_0_axi_periph
S00_AXI
ACLK
ARESETN[0:0]
S00_ACLK
S00_ARESETN[0:0]
M00_ACLK
M00_ARESETN[0:0]
M01_ACLK
M01_ARESETN[0:0]
M02_ACLK
M02_ARESETN
M00_AXI
M01_AXI
M02_AXI
AXI Interconnect
rst_processing_system7_0_50M
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
processing_system7_0
M_AXI_GP0_ACLK
ZYNQ
DDR
FIXED_IO
M_AXI_GP0
FCLK_CLK0
FCLK_RESET0_N
ZYNQ7 Processing System
DDR
FIXED_IO

My IP – Memory Address Range

Diagram | **Address Editor**

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

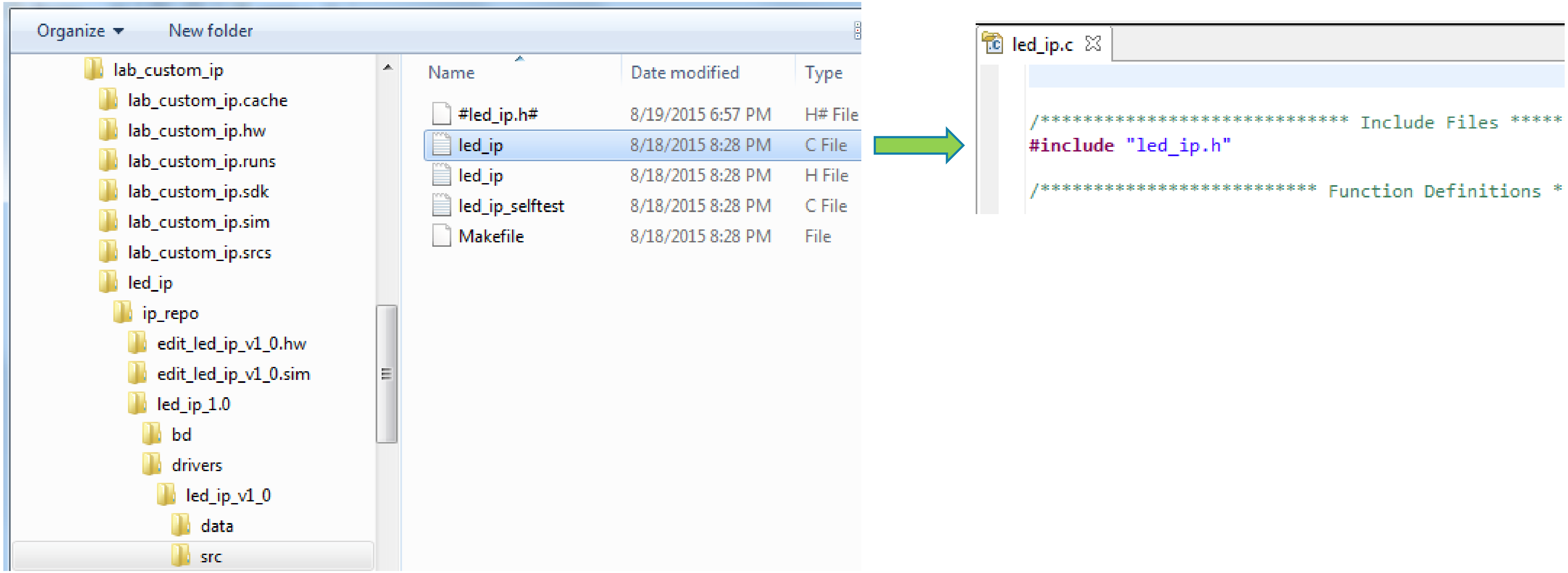
- The *driver code* are generated automatically when the IP template is created.
- The *driver* includes higher level functions which can be called from the user application.
- The *driver* will implement the low level functionality used to control your peripheral.

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(…)*

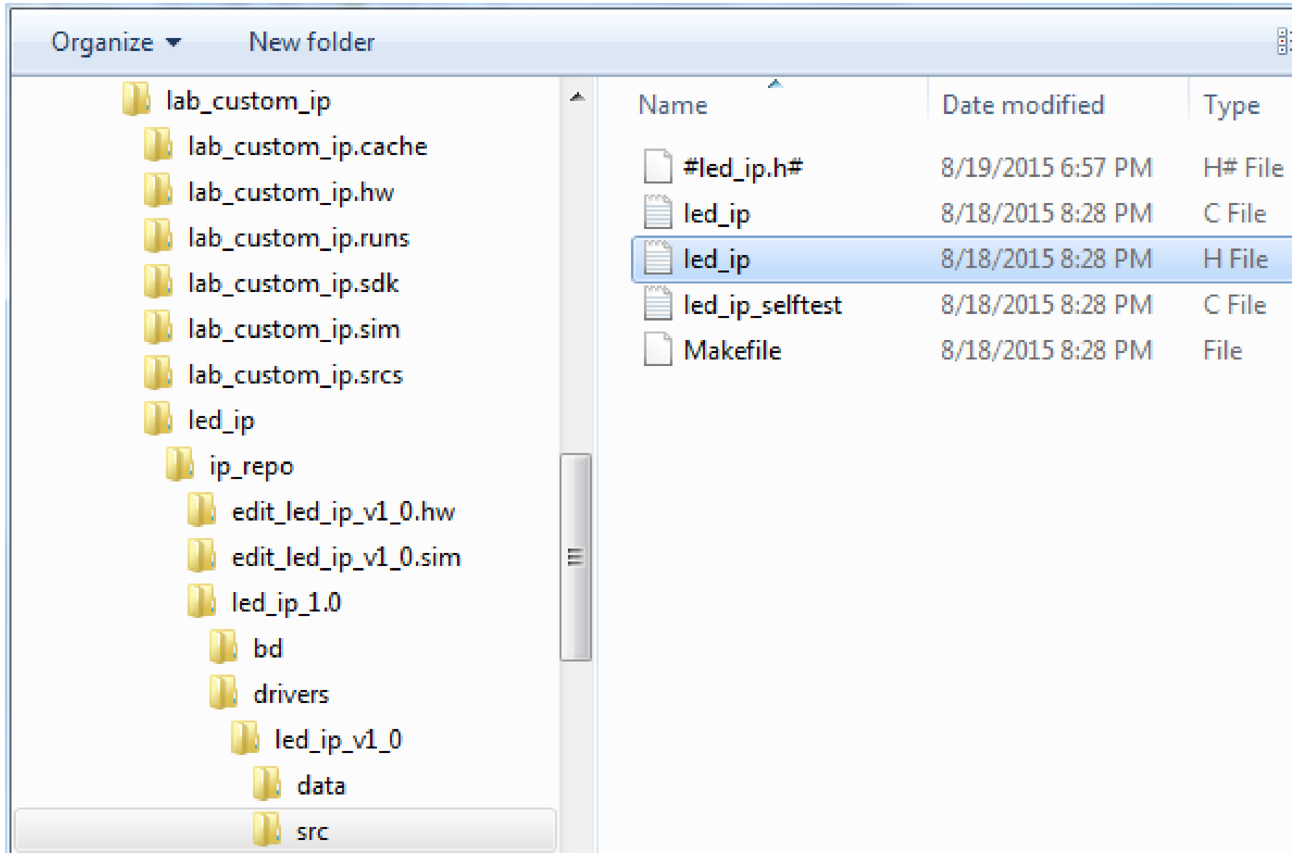
Custom IP Drivers: *.c

led_ip\ip_repo\led_ip_1.0\drivers\led_ip_v1_0\src\led_ip.c



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

led_ip\ip_repo\led_ip_1.0\drivers\led_ip_v1_0\src\led_ip.h

led_ip.h

```c
/****************** 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

led_ip\ip_repo\led_ip_1.0\drivers\led_ip_v1_0\src\led_ip.h

```c
/**
 *
 * 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

led_ip\ip_repo\led_ip_1.0\drivers\led_ip_v1_0\src\led_ip.h

```c
/**
 *
 * 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

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

```c
#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);



      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)*

```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;
}
```

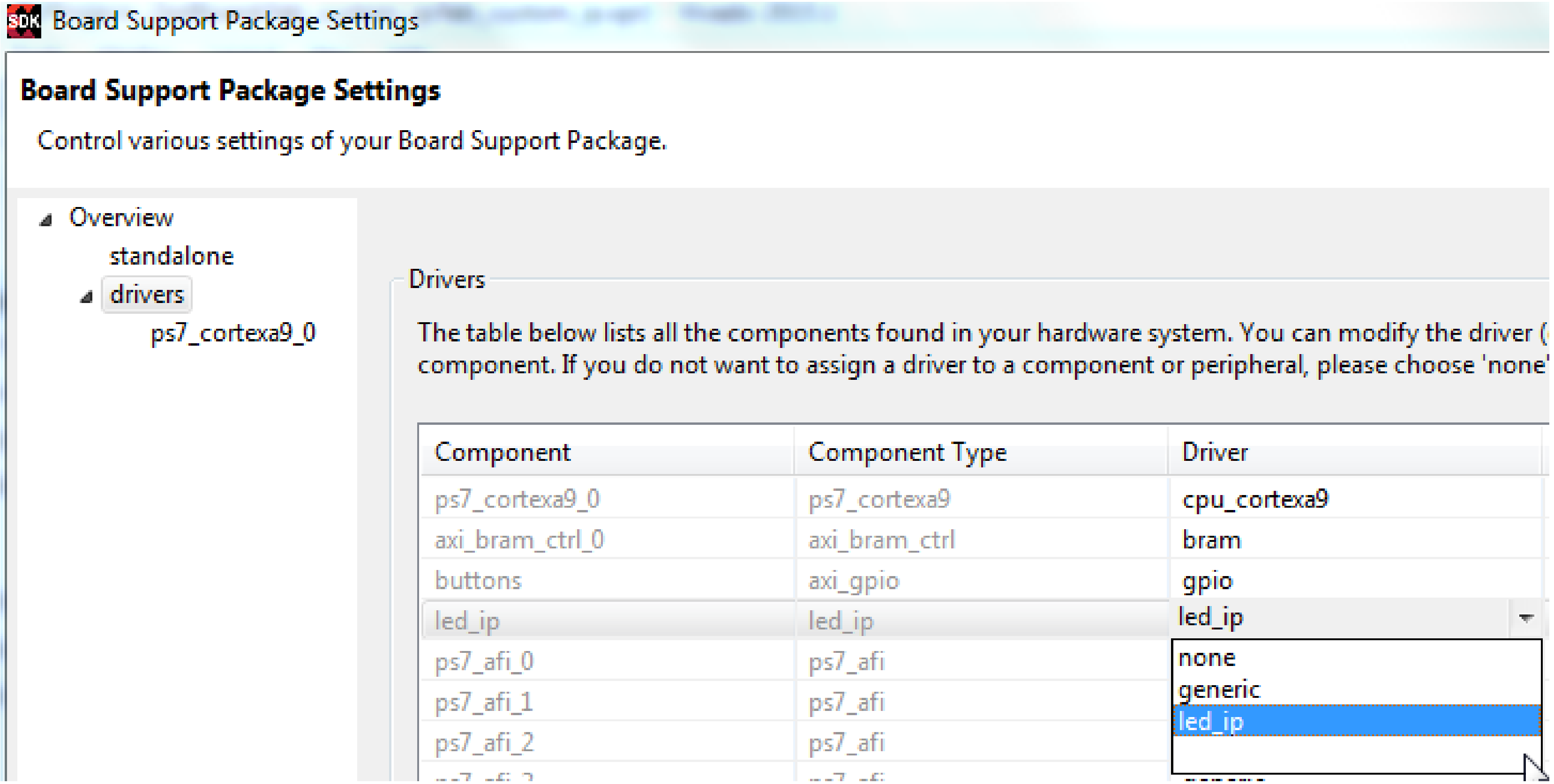
IP Drivers – *Xil_Out32 (xil_io.h/xil_io.c)*

```
/******************************************************************************/
/**
* 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 – Vitis ‘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 – Vitis ‘Activation’



System Level Address Map

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters[1]	Notes
0000_0000 to 0003_FFFF[2]	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[3]
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 Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF[4]	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF[2]	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

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

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

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

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

Macro

Bibliography

- “*Introducción a la Programación en Lenguaje C para Ingeniería Electrónica*”, S. Burgos, Omar Berardi. Dictumediciones, 2015.
- Xilinx Standard C Libraries
- “*Standalone Library Documentation BSP and Libraries Document Collection*”. AMD UG643 (V2025.1).