Embedded 'C' for Zynq-SoPC

Cristian Sisterna

Universidad Nacional San Juan

Argentina

Embedded C

Embedded C Programming Language, which is widely used in the development of Embedded Systems, is an extension of C Program Language

The extension in ***Embedded C*** from standard *C Programming Language* include I/O Hardware Addressing, fixed point arithmetic operations, accessing address spaces, etc.

Standard C and Embedded C: Differences (1)

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**

Standard C is general-purpose programming language used for developing any type of desktop applications

Embedded C, that is an extension of Standard C, is used in the development of microcontroller/microprocessor based applications

Standard C and Embedded C: Differences (2)

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 systems 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 controlled by the processing power, timing constraints. Whereas code size depends on the available program memory

Standard C and Embedded C: Differences (3)

Compilers for ***C*** (ANSI C) typically generate OS dependent executables. *Embedded C* is OS independent

Embedded C requires specific compilers to create files to be downloaded to the microcontrollers/microprocessors where it needs to run

Embedded C is fully hardware dependent language

OS system is a must for C programming whereas it’s an option for ***Embedded C***

Advantages of Using *Embedded C*

- It is small and reasonably simpler to learn, understand, program and debug
- Embedded C Compilers are available for almost all embedded devices in use today, and there is a large pool of experienced C programmers
- It supports easy access to I/O, memory map and facilitates the management of large embedded projects
- Unlike assembly, Embedded C has advantage of processor-independence and is not specific to any particular microprocessor/ microcontroller or any system

Basic Embedded ‘C’ Program Template

Basic Embedded Program Architecture

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

Executed once

Infinite loop

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

Embedded C - Basic Example

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

Tasks for the infinite loop ????

1. reading the interval value from the switches

2. toggling the two LEDs after a specific amount of time

Embedded C - 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 10 switches
* argument:
*      switch-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
}
```

Basic Example - Writing

```
/*******************************************************************************************
* function: led.flash ()
* purpose: toggle 2 LEDs according to the given period
* argument:
*         addr_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
}
```

Basic Example – Read / Write

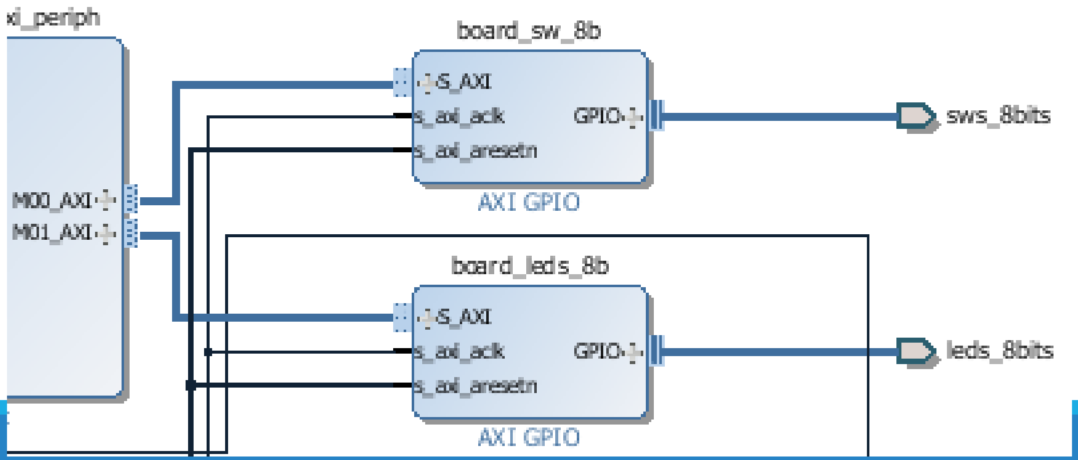
```c
int main()
  {
  int period;

  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;
}
```

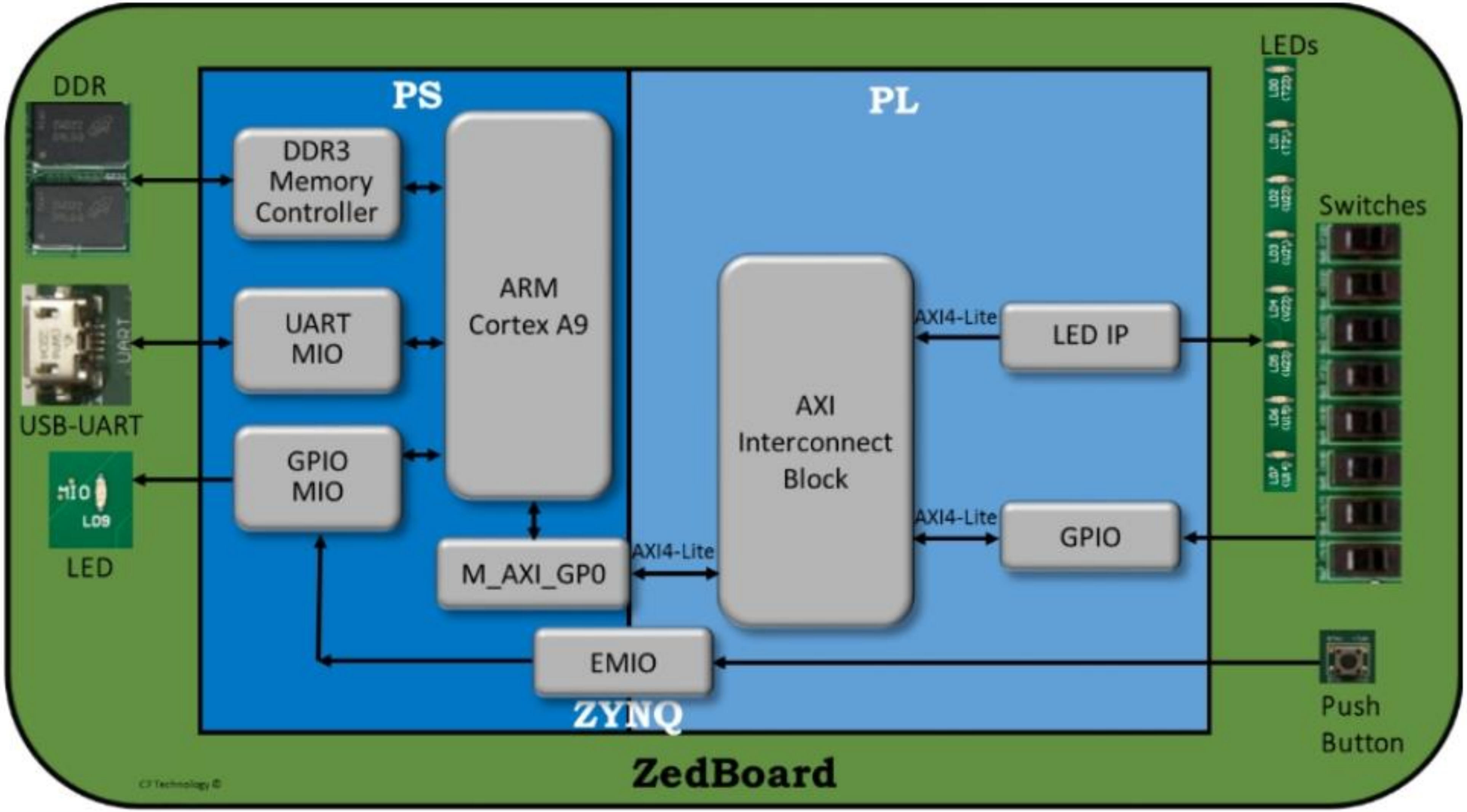
```c
void led_flash(u32 addr_led_base, int period)
{
 static u8 led_pattern = 0x01;
 unsigned long i, itr;          //static?
  led_pattern ^= 0x03;
  my_iowr(addr_led_base, led_pattern);
  itr = period * 2500;
  for (i=0; i<itr; i++) {}
}
```

Read/Write From/To GPIO Inputs and Outputs

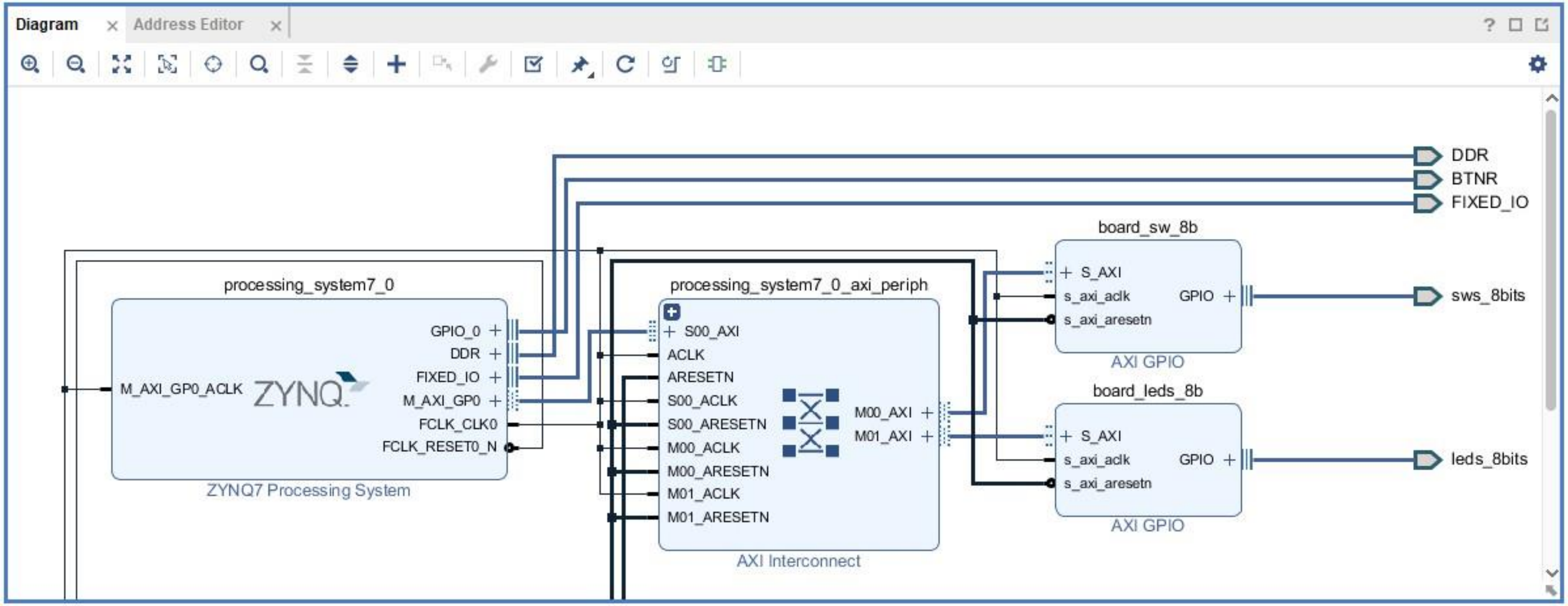
Zynq SoPC



Read/Write From/To GPIO Inputs and Outputs



Read/Write From/To GPIO Inputs and Outputs



Steps for Data Reading from a GPIO – Embedded C

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**

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

int main (void)
 {
   XGpio switches;
   XGpio leds;
   . . .
```

```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;
```

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

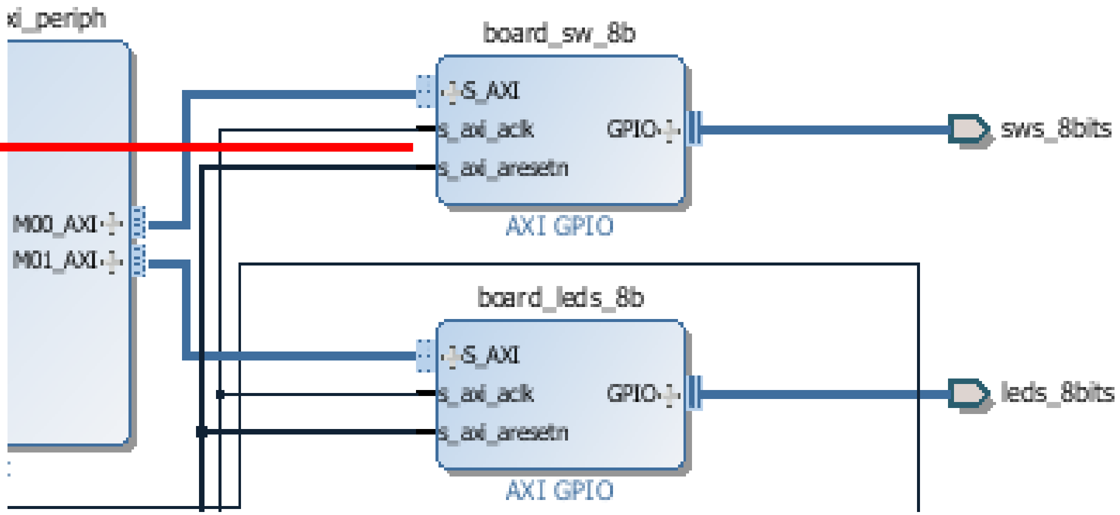
Steps for Reading from a GPIO – Step 2(cont')

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

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

xparameters.h | lab_gpio_in_out.c

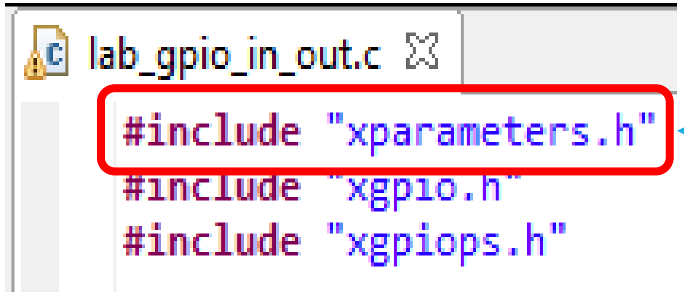
```
/* Definitions for peripheral BOARD_SW_8B */
#define XPAR_BOARD_SW_8B_BASEADDR 0x41200000
#define XPAR_BOARD_SW_8B_HIGHADDR 0x4120FFFF
#define XPAR_BOARD_SW_8B_DEVICE_ID 1
#define XPAR_BOARD_SW_8B_INTERRUPT_PRESENT 0
#define XPAR_BOARD_SW_8B_IS_DUAL 0
```



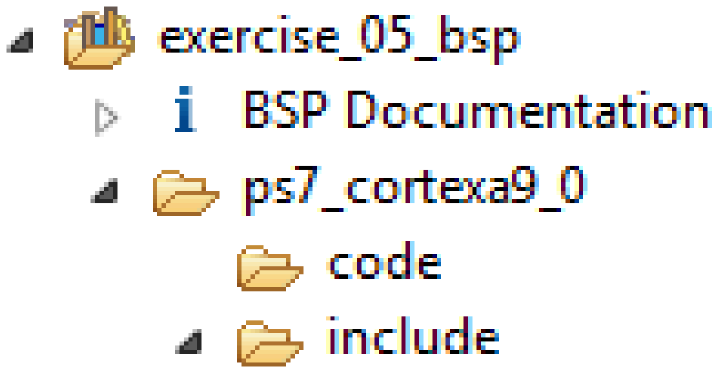
xparameters.h

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

This file is generated from the hardware platform created in Vivado

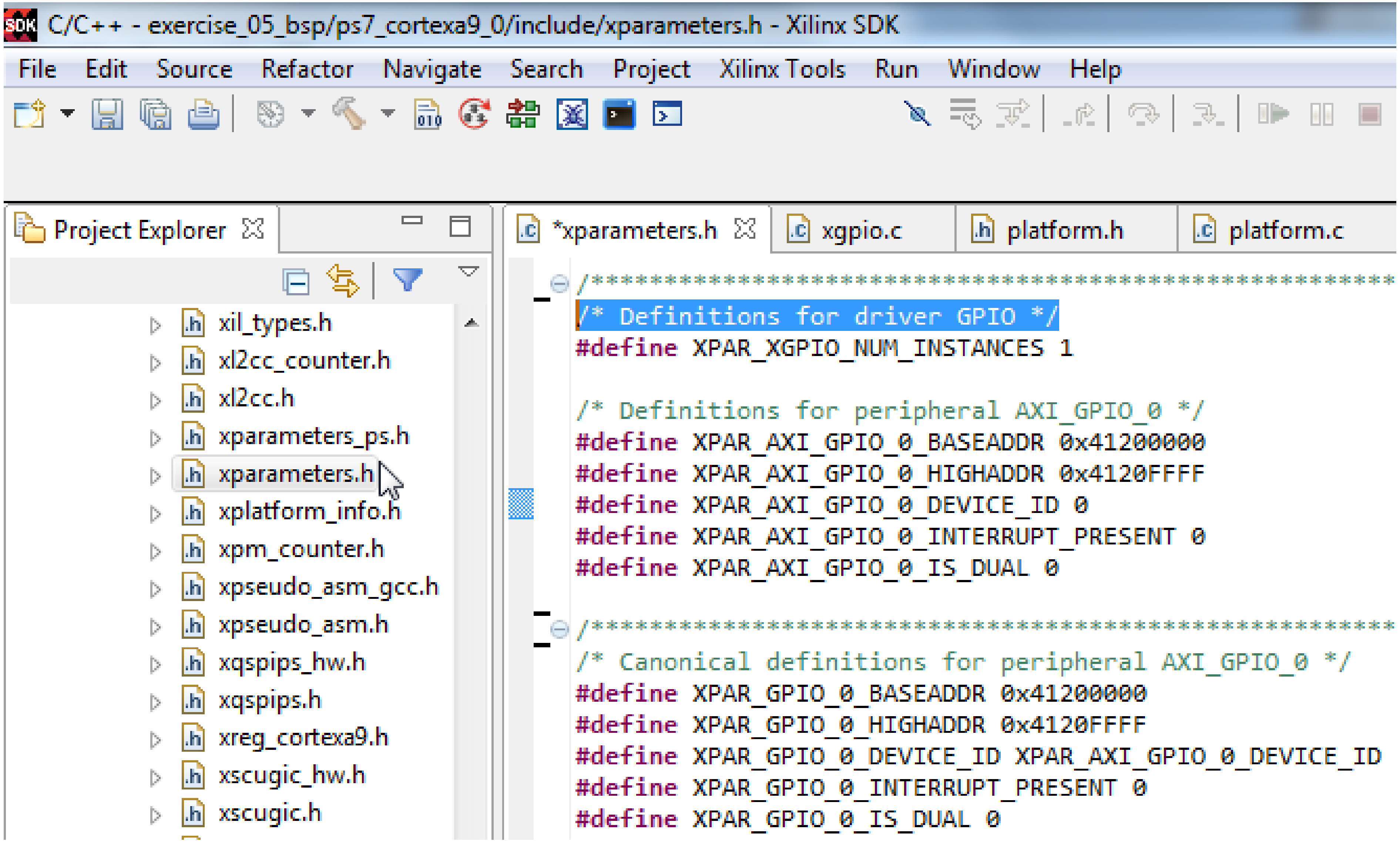


Ctrl + Mouse Over



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.

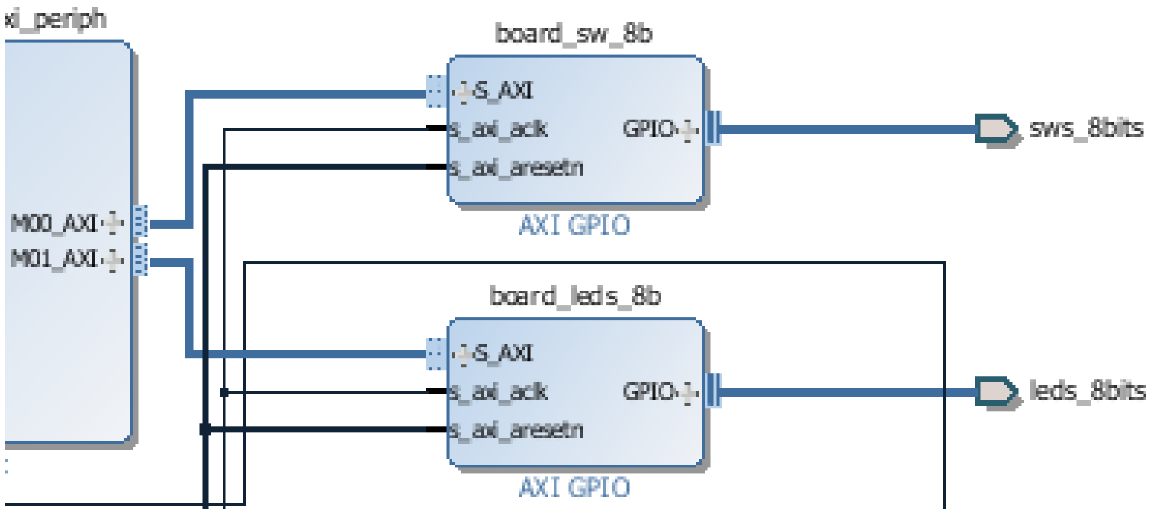
DirectionMask: is a bitmask specifying which bits are inputs and which are outputs. Bits set to **‘0’ are output**, bits set to **‘1’ are inputs**.

Return: none

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

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 (cont')

```
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

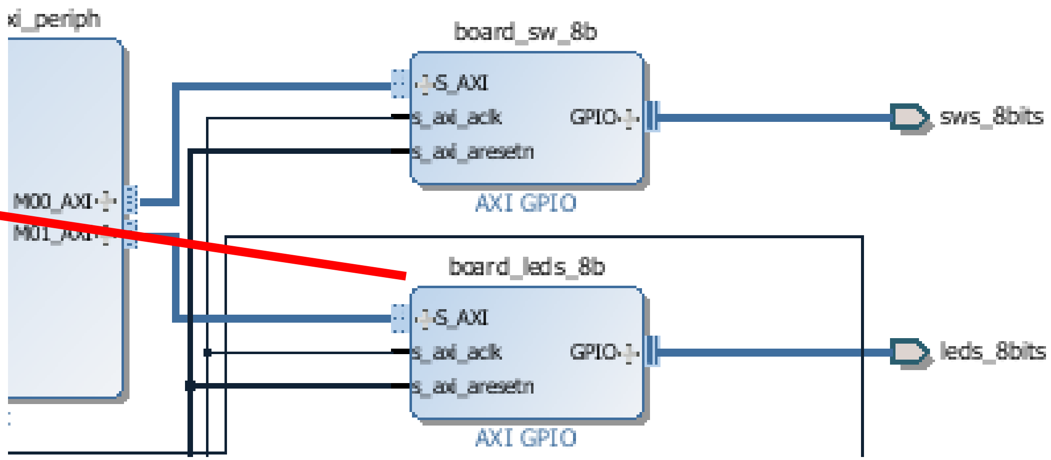
Steps for Writing to a GPIO – Step 2(cont')

```
(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

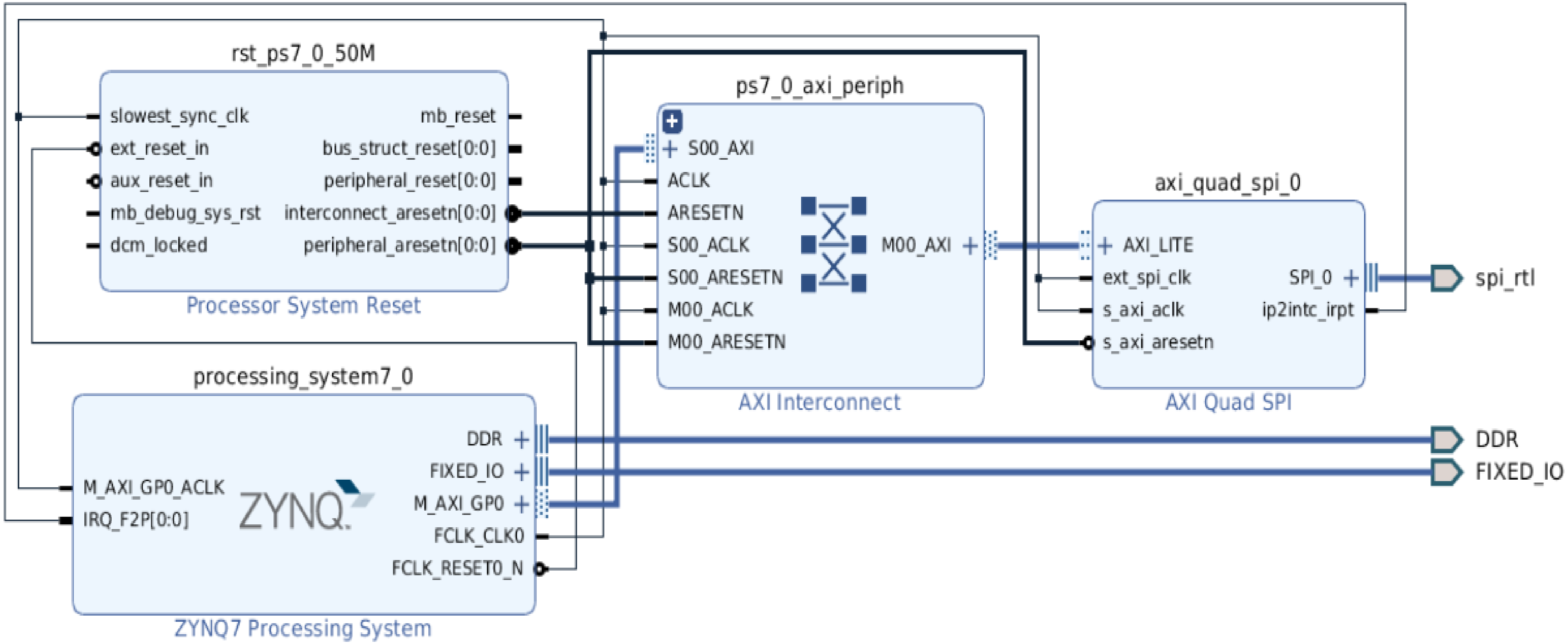
Steps for Writing to a GPIO – Step 3 (cont')

```
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);
```

‘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 – Driver Example

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

Appendix

Reviewing Embedded ‘C’ Basic Concepts

Embedded C – Xilinx ‘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*

Global Variables

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

Global and Local Variables Declarations

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

Global Variables

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

int function1()
{
int alarm = 128;
```

Local Variable

```
     ...
     alarm =+1;
     flag  = 3;
     ...
}
```

Local Variables

- 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 compiler will generally use the extended addressing mode to access *global variables* or indexed addressing mode if they are accessed though a pointer

Use of the '*static*' modifier

- 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 '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)

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

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

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

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.

Volatile Variable

```c
#include <stdio.h>

/* Optimization code snippet 1 */
#include<stdio.h>

int x = 0;

int main()
{
    if (x == 0) // This condition is always
    {
        printf(" x = 0 \n");
    }
    else        // Else part will be optimiz
    {
        printf(" x != 0 \n");
    }
    return 0;
}
```

```c
#include<stdio.h>

volatile int = 0;     /* volatile Keyword*/

int main()
{
    x = 0;

    if (x == 0)
    {
    printf(" x = 0 \n");
    }
    else        // Now compiler never optimize else part because the
    {           // variable is declared as volatile
    printf(" x != 0 \n");
    }
    return 0;
}
```

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);

/**
* Initialize the XGpio instance provided by the caller based on the
* given DeviceID.
*
* 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

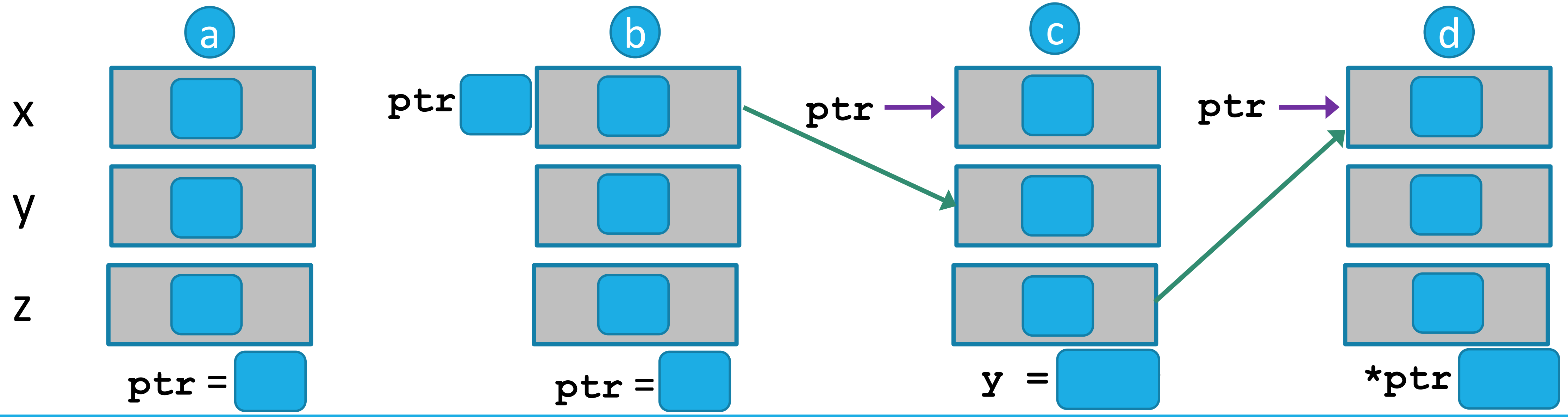
```c
#include "xparameters.h"
#include "xgpio.h"
#include "xgpiops.h"

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

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

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

```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; // y gets content pointed by ptr
d  *ptr = z;    // location pointed by ptr gets the value 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);
                }
        }
}
```

Data Unpacking

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

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

```
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 ));
```

Data Packing

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 chl, ch0;
```

Packing

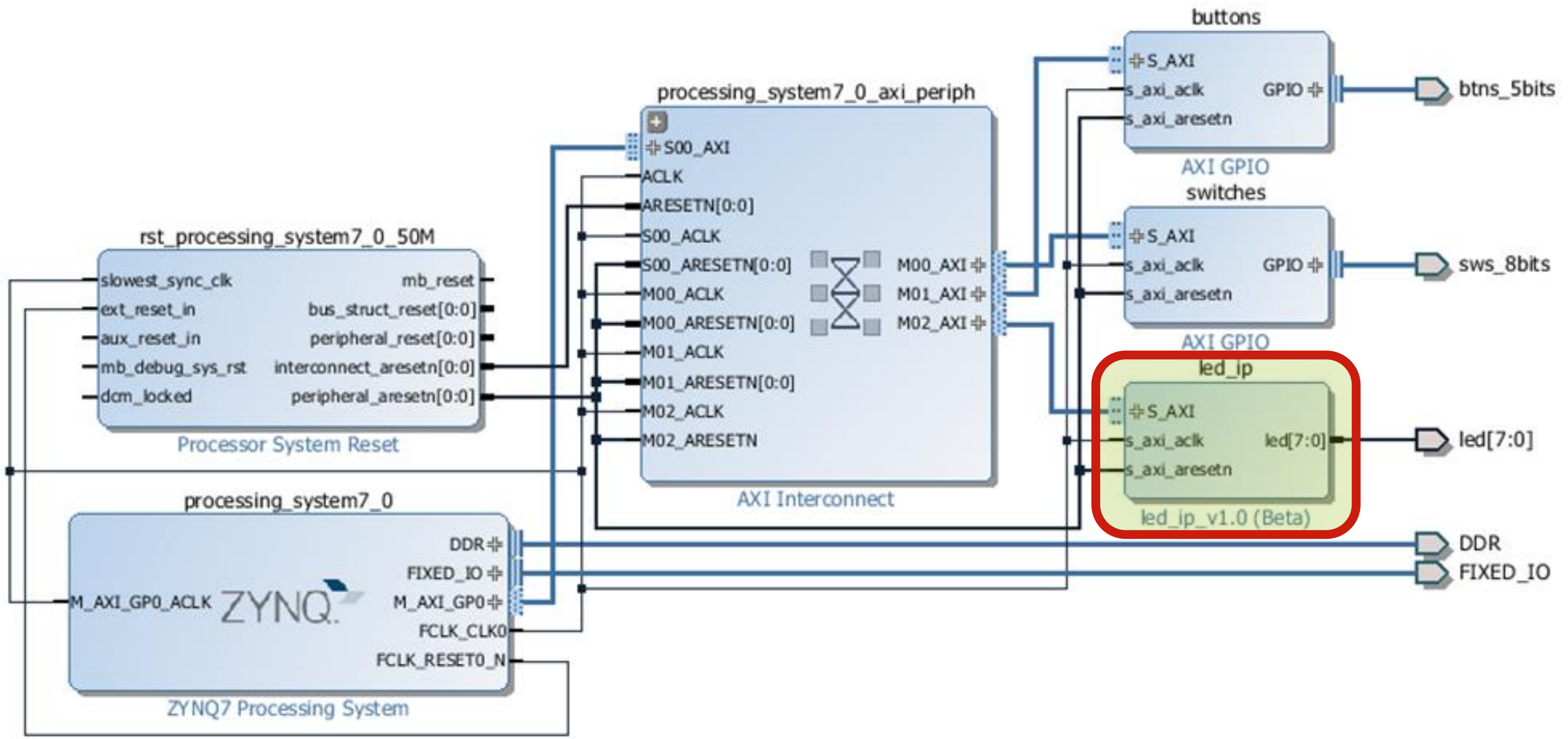
```
wr_data = (u32)(num);                          //num[15:0]
wr_data = (wr_data << 8) | (u32) ch1; //num[23:8],ch1[7:0]
wr_data = (wr_data << 8) | (u32) ch0; //num[31:16],ch1[15:8]
my_iowr( . . . , wr_data) ;                   //ch0[7:0]
```

Another Way of Data Packing….

```
wr_data = (((u32)(num))<<16)|(((u32)ch1)<<8)|(u32)ch0;
```

‘C’ Drivers for Custom IP

Custom IP



buttons
S_AXI
s_axi_aclk
s_axi_aresetn
GPIO
AXI GPIO
btns_5bits
switches
S_AXI
s_axi_aclk
s_axi_aresetn
GPIO
AXI GPIO
sws_8bits
led_ip
S_AXI
s_axi_aclk
s_axi_aresetn
led[7:0]
led_ip_v1.0 (Beta)
led[7:0]
DDR
FIXED_IO
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

My IP – Memory Address Range

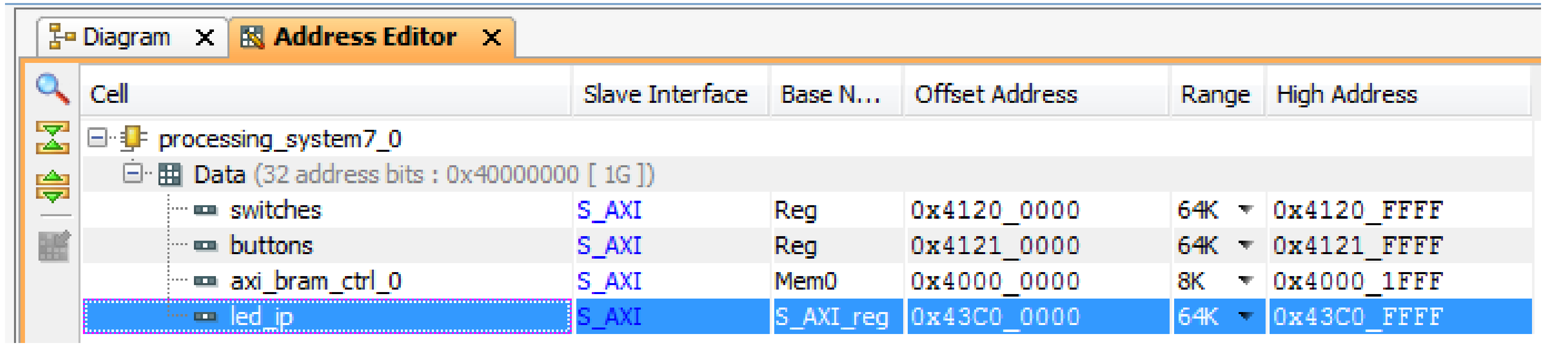


Diagram | Address Editor

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

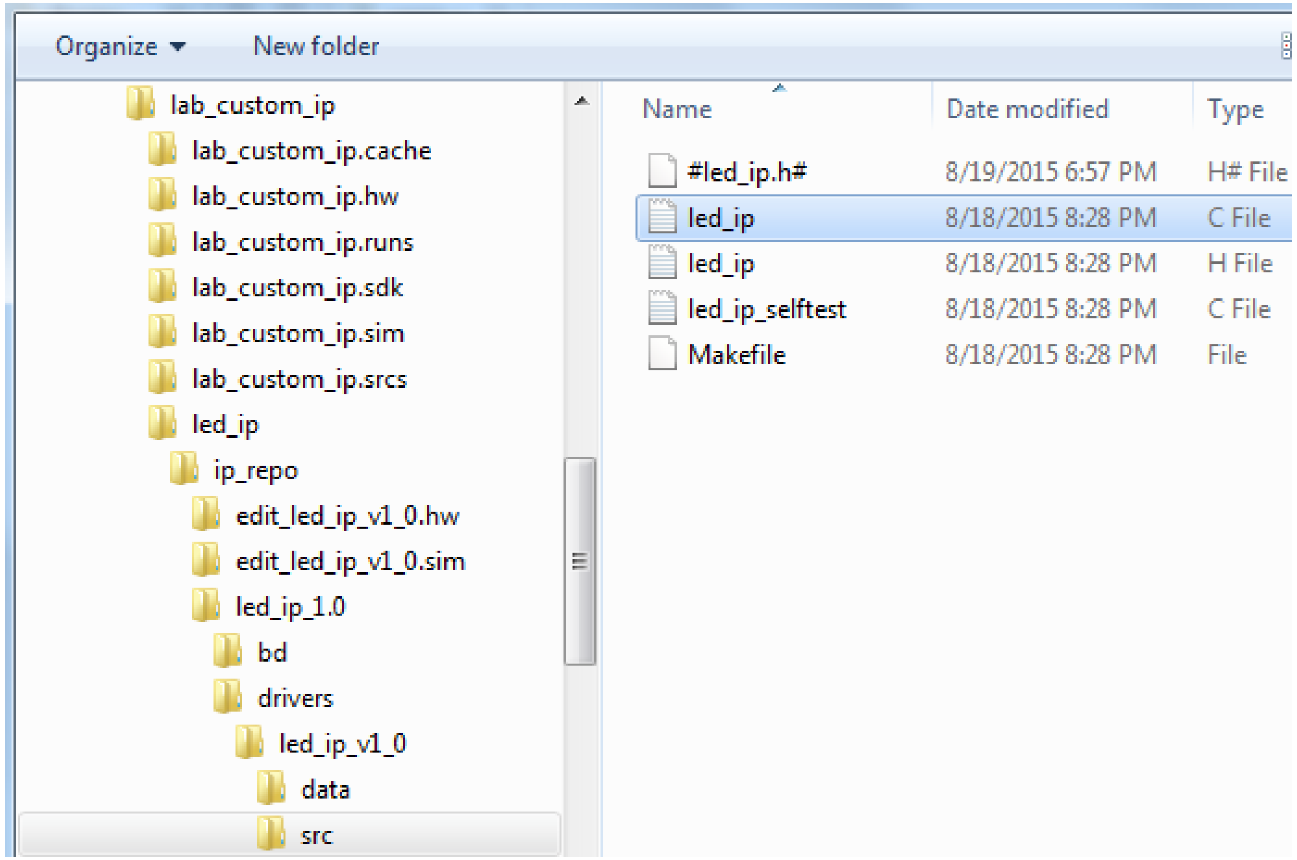
- The *driver code* are generated automatically when the IP template is created.
- The *driver* includes higher level functions which can be called from the user application.
- The *driver* will implement the low level functionality used to control your peripheral.

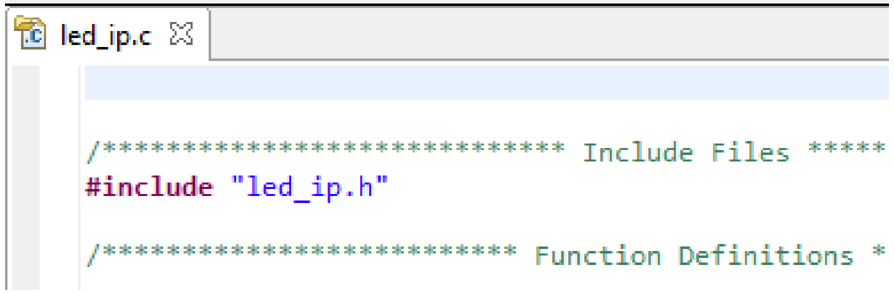
led_ip\ip_repo\led_ip_1.0\drivers\led_ip_v1_0\src

- *led_ip.c*
- *led_ip.h*
  - *LED_IP_mWriteReg(…)*
  - *LED_IP_mReadReg(…)*

Custom IP Drivers: *.c

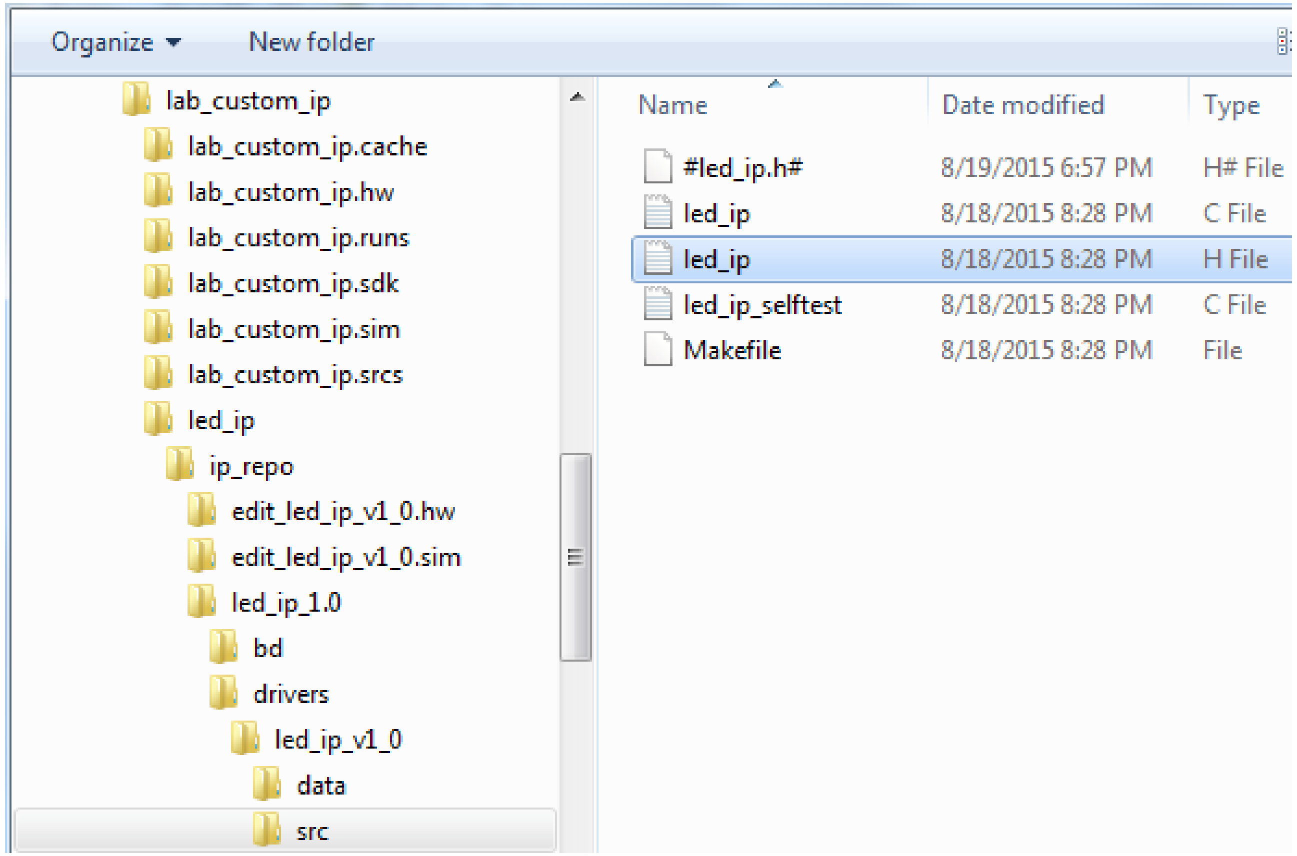
led_ip\ip_repo\led_ip_1.0\drivers\led_ip_v1_0\src\led_ip.c





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 (cont' 1)

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

led_ip.h

```
/****************** 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

```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)*

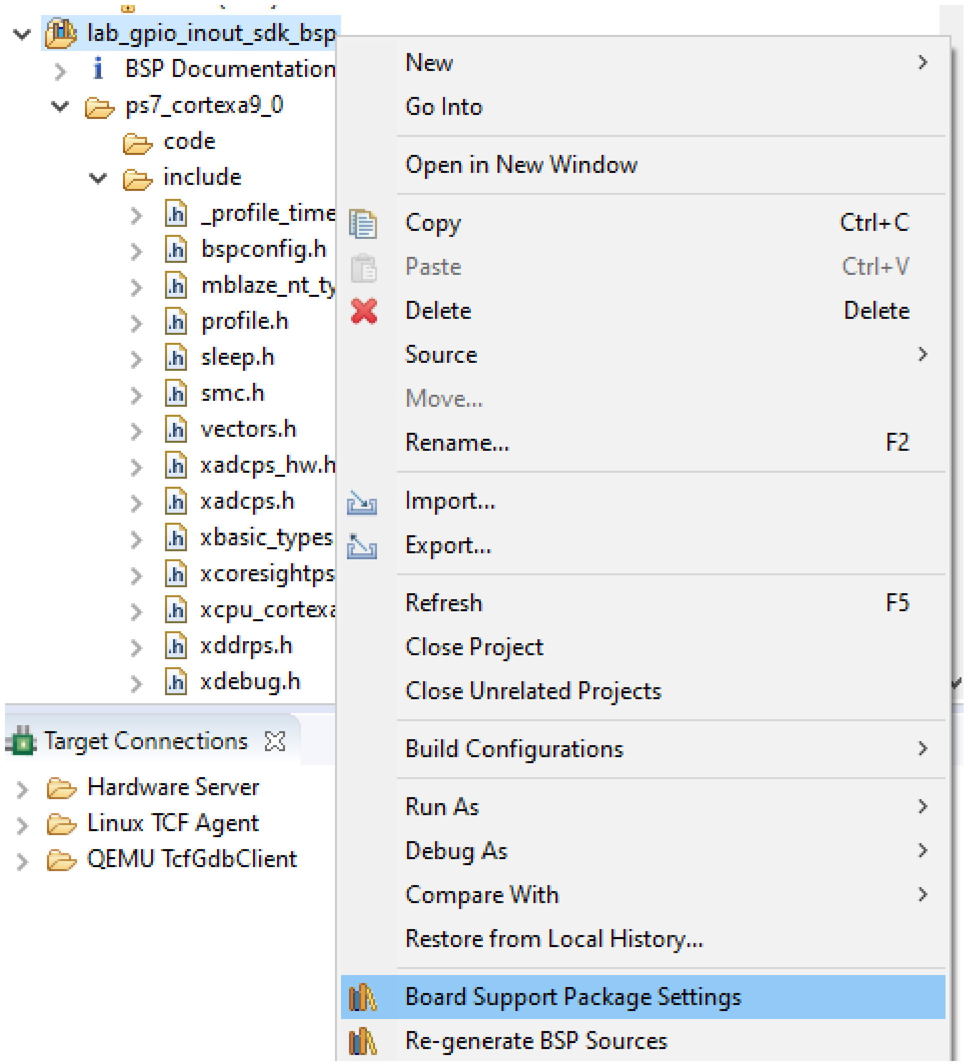
```
/******************************************************************************/
/**
* Performs an input operation for a 32-bit memory location by reading from the
* specified address and returning the Value read from that address.
*
* @param      Addr contains the address to perform the input operation 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;
}
```

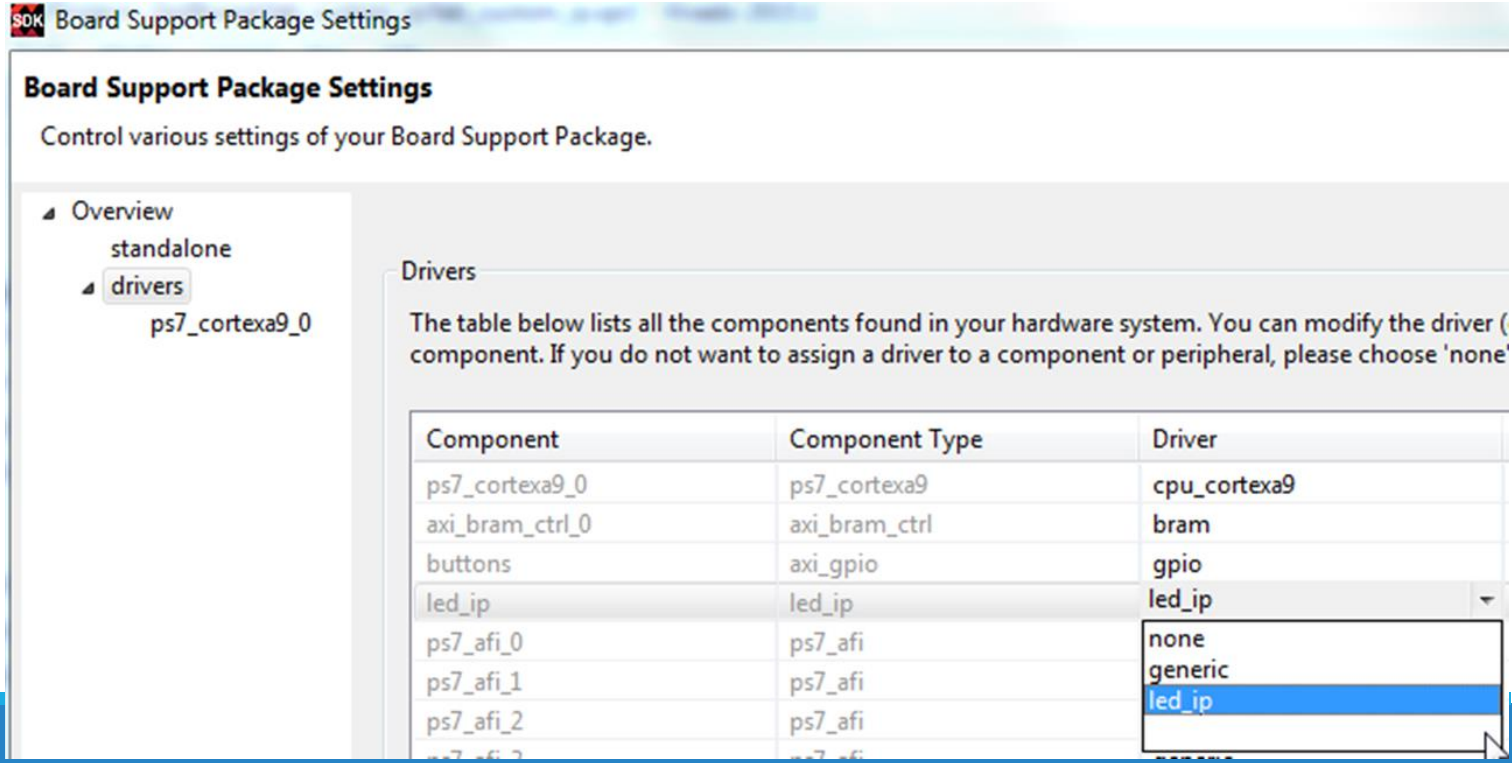
IP Drivers – SDK 'Activation'

- Select <***project_name***>***_bsp*** in the project view pane. Right-click
- Select ***Board Support Package Settings***



IP Drivers – SDK 'Activation'

- 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



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