LABORATORY 5

Zynq Design

Debugging

Debugging a Zynq Embedded System

Introduction

In this exercise, we will look at the concepts behind debugging a piece of ‘C’ code. Debugging is an important process in the software development flow. There will be cases in which you may not need to debug, but those will be the least cases. The SDK comes with a very advanced debug and trace interface. The purpose of this lab is to learn some basic debugging techniques.

Objectives

After completing this lab, you will be able to:

- Know the different debugging tool offered by SDK
- Know which are the different perspective in the SDK environment
- Configure breakpoints in the ‘C’ code
- Know how to look at variables defined in the ‘C’ code, and how to modify its value
- Know how to look at specific memory addresses.
- Which is the best approach to debug your design

Procedure

The embedded system built in the lab4 will be used to get familiar with the debug process and its main features.

Create a Copy of a Vivado Project *Step 1*

1-1. ***Objective*: Launch Vivado, open *lab4* and create a copy of it.**

1-2. Start *Vivado*.

1-3. Click *Open Project*, and open *lab4* (*lab_gpio_in_out*)

1-4. In the *Vivado* environment, use the option ***File->Save Project As*** and use *lab5* or *lab_debugging* as project name. Use the directory that you like.

After *Finish* is clicked, the *New Project* wizard closes and the project you just created opens in *Vivado GUI.*

Lunch SDK and Execute "Debug as…" *Step 2*

2-1. ***Objective*: From Vivado generate the bitstream, export hardware and lunch SDK. In SDK execute the "Debug as" option to open the debug perspective in SDK. Use of the different debug tools.**

2-1-1. Since in step 1 of this lab we saved an old project with an new name it could happens that *Vivado* indicates that the project is out-of-date, therefore it will be necessary to generate the *bitstream* again. Hence, following the steps explained in previous labs, generate the bitstream for this project.

2-1-2. Export the hardware to SDK by doing: *File -> Export -> Export Hardware* (remember to check the "*Include Bitstream*" option).

2-1-3. Do *File -> Lunch SDK***.**

2-1-4. Follow the same steps explained in *Lab4* to create an application project in SDK**.** Name the application project: ***lab_debug***.

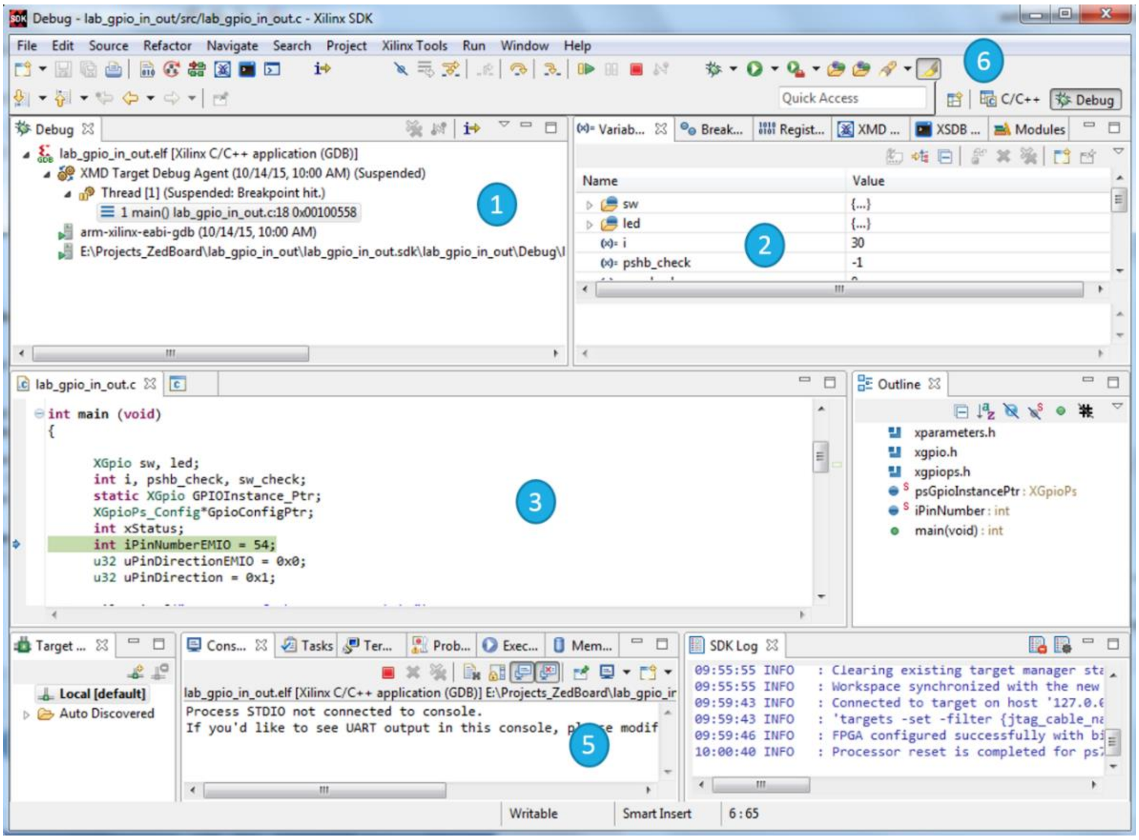
2-1-5. Import the 'C' code you use in ***Lab4***.

2-1-6. Configure the *PL* part, using either *Xilinx Tools -> Program FPGA* or the icon. Wait for the blue light to turn on.

2-1-7. Right click on the name you selected for the SDK application, and choose "*Debug As -> Lunch on Hardware (System Debugger)".*

2-1-8. After a few seconds, you may see a pop-up message asking you whether you want to switch to the "*Debug Perspective"*. Choose **"***Yes*".

2-1-9. The SDK windows will re-arrange themselves, and you will now be in the "*Debug Perspective*" rather than the "*C/C++ Perspective*". Following figure shows the "*Debug Perspective*" (a "Perspective" is the name given to a specific layout of various panes on the SDK main window).



Debug - lab_gpio_in_out/src/lab_gpio_in_out.c - Xilinx SDK

Debug

- lab_gpio_in_out.elf [Xilinx C/C++ application (GDB)]
  - XMD Target Debug Agent (10/14/15, 10:00 AM) (Suspended)
    - Thread [1] (Suspended: Breakpoint hit.)
      - 1 main() lab_gpio_in_out.c:18 0x00100558
  - arm-xilinx-eabi-gdb (10/14/15, 10:00 AM)
  - E:\Projects_ZedBoard\lab_gpio_in_out\lab_gpio_in_out.sdk\lab_gpio_in_out\Debug\l

Variab… | Break… | Regist… | XMD … | XSDB … | Modules

Name	Value
sw	{…}
led	{…}
i	30
pshb_check	-1

lab_gpio_in_out.c

```
int main (void)
{

    XGpio sw, led;
    int i, pshb_check, sw_check;
    static XGpio GPIOInstance_Ptr;
    XGpioPs_Config*GpioConfigPtr;
    int xStatus;
    int iPinNumberEMIO = 54;
    u32 uPinDirectionEMIO = 0x0;
    u32 uPinDirection = 0x1;
```

Outline

- xparameters.h
- xgpio.h
- xgpiops.h
- psGpioInstancePtr : XGpioPs
- iPinNumber : int
- main(void) : int

Target … : Local [default], Auto Discovered

Cons… | Tasks | Ter… | Prob… | Exec… | Mem…

```
lab_gpio_in_out.elf [Xilinx C/C++ application (GDB)] E:\Projects_ZedBoard\lab_gpio_ir
Process STDIO not connected to console.
If you'd like to see UART output in this console, pl   e modif
```

SDK Log

```
09:55:55 INFO   : Clearing existing target manager sta
09:55:55 INFO   : Workspace synchronized with the new
09:59:43 INFO   : Connected to target on host '127.0.0
09:59:43 INFO   : 'targets -set -filter {jtag_cable_na
09:59:46 INFO   : FPGA configured successfully with bi
10:00:40 INFO   : Processor reset is completed for ps7
```

Writable | Smart Insert | 6 : 65

The different parts that make up the debug perspective are:

1- Stack frame for target threads

2- Variables, breakpoints and registers views

3- C / C++ editor

4- Code outline and disassembly view

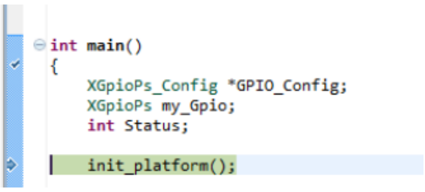
5- Messages and memory view (when enabled)

6- Debug tool bar

Note: advanced users can even design and use their own perspectives, to match their precise needs and personal preferences.

2-1-10. With the debug perspective open, you will notice that the code-editing window (3) has become smaller and has moved to the left part of the screen. The Project Explorer window has vanished altogether; there is no need to concentrate on different applications because we are debugging just one.

In the code editing window in the Debug perspective you will find out that there is a line of the code that is highlighted in green, and also marked with a blue arrow in the left margin.



```
int main()
{
    XGpioPs_Config *GPIO_Config;
    XGpioPs my_Gpio;
    int Status;

    init_platform();
```

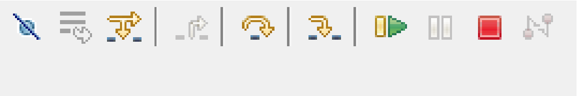
This is showing you that the processor has been halted at this instruction of the code, and that the software execution is under control of the GDB debugger (**G**NU **De**bugger). The highlighted line of code is telling you that is waiting for a command (next step in this description) to see when or how to execute this instruction.

The pane above the code editing window 1 shows you that the *lab_debug.elf* file is being debugged.

Note: the debug control is handled by the Target Communication Framework, TCF, system. It manages the connection between the GDB debug engine, and the JTAG cable. You will be able to see that the code is halted inside the "main" function. This display will be updated to show when you are inside a function, and the hierarchy will be shown to demonstrate which functions called each other as you step through the code.

2-1-11. Let's use the basic debug tools available in the debug perspective in the SDK environment.

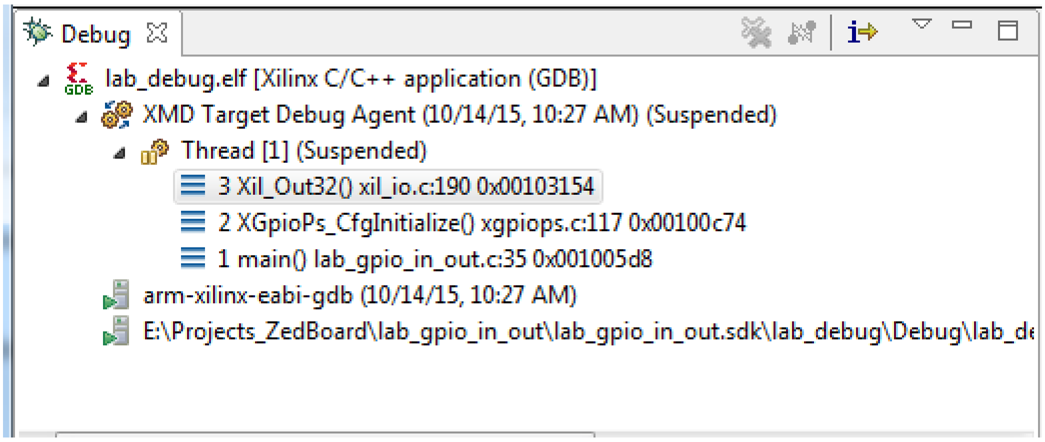
In the Debug Tool Bar 6 there are a series of icons that represent different debug processes.



- ▶ "***Resume***": a green arrow button which will execute the code until the end of the code or until it reaches the next breakpoint.
- 🟥 "***Terminate***": a square red button to halt execution completely wherever you are in the application code.

- "***Step Into***": a yellow arrow that, when pressed, will continue the execution of the code from inside of the function you are halt in. It allows go inside any function.
- "***Step Over***": a yellow arrow that will execute the function without going inside of it.
- "***Step Return***": a yellow arrow that will allow to exit from inside a function that you get into using "***Step Into***".

2-1-12. Getting back to the debug process, click on the *"Step Over"* icon until you get to the "`xStatus = XGpioPs_CfgInitialize….`" function. Now click on the "*Step Into*" icon. You are now inside the `XGpioPs_CfgInitialize` function. Then use the "*Step Over*" until you get to the `XGpioPs_WriteReg` function. Then use "*Step Into*", you will see that the next function called is `Xil_Out32`. You should also see that the "*Debug*" is being updated on each "*Step Into*" click. Here is a screenshot of the *Debug* pane:



```
lab_debug.elf [Xilinx C/C++ application (GDB)]
    XMD Target Debug Agent (10/14/15, 10:27 AM) (Suspended)
        Thread [1] (Suspended)
            3 Xil_Out32() xil_io.c:190 0x00103154
            2 XGpioPs_CfgInitialize() xgpiops.c:117 0x00100c74
            1 main() lab_gpio_in_out.c:35 0x001005d8
    arm-xilinx-eabi-gdb (10/14/15, 10:27 AM)
    E:\Projects_ZedBoard\lab_gpio_in_out\lab_gpio_in_out.sdk\lab_debug\Debug\lab_de
```

The Debug pane is saying that the *`main()`* function is the higher hierarchy function. Underneath *`main()`*, there is *`XGpioPs_CfgInitialize`*, and then we have *`Xil_Out32`*, that is underneath of *`XGpioPs_CfgInitialize.`* This information is very useful specially when there are several function calls, and mainly to set up a breakpoint in the correct function.

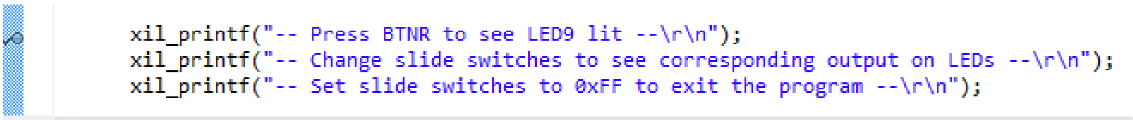
2-1-13. Now use the "*Step Return*" button as many times as necessary to return the *`main()`* function. By clicking this button, all the instructions to the end of the current function are executed and then goes automatically one level above.

Setting Up a Breakpoint

2-1-14. In the code editing pane, scroll down a few lines and find the line of code which print "-- Press BTNR to see LED9 lit –".

To the left of this line, in the blue margin, do a double click on your mouse and it should appear a blue dot with a tick mark.

You have set a *breakpoint*, and tells GDB that you wish to stop at this function.

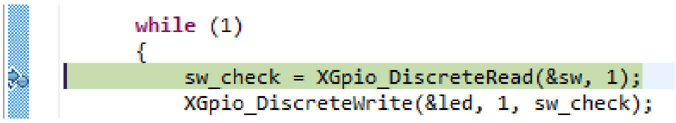


```
    xil_printf("-- Press BTNR to see LED9 lit --\r\n");
    xil_printf("-- Change slide switches to see corresponding output on LEDs --\r\n");
    xil_printf("-- Set slide switches to 0xFF to exit the program --\r\n");
```

2-1-15. Now click the "resume" button (green arrow), and the debugger will execute all of your code from the current position to the breakpoint that you have just set (you will notice that the breakpoint line is highlighted green, it means this is the current statement in the debug process).

This is a very useful way to execute a large part of your application, but only stop at the section you wish to debug.

2-1-16. Now you execute as many "*Step Over*" as needed until reach the code line "`sw_check = XGpio_DiscreteRead(&sw, 1);`".

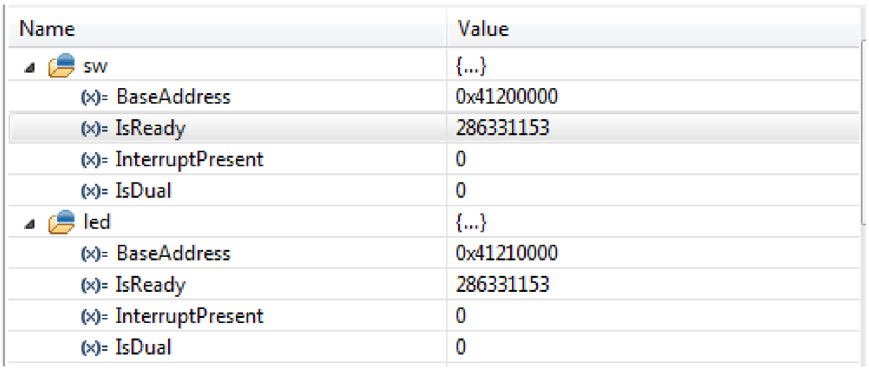


```
    while (1)
    {
        sw_check = XGpio_DiscreteRead(&sw, 1);
        XGpio_DiscreteWrite(&led, 1, sw_check);
```

At this stage we will turn our attention to the *Variables pane* 2.

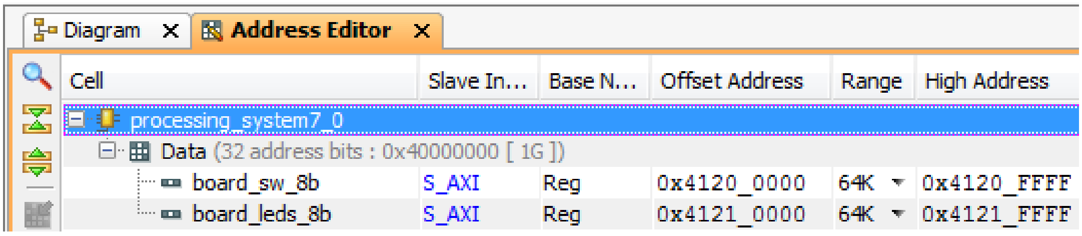
In this pane, you will see all the local variables, which are relative to the function you are halt on. At this time you are in the *main()* function; therefore all the variables of the main function are displayed.

In the "*Variables*" pane you should also see the defined structures, such as `sw` and `led`. Remember that a structure is a collection of variables under one name. By default, the *BaseAddress* is showed in decimal format, for our understanding is better to read that address in hexadecimal. Hence, right click over the decimal value and select *Format -> Hexadecimal*.



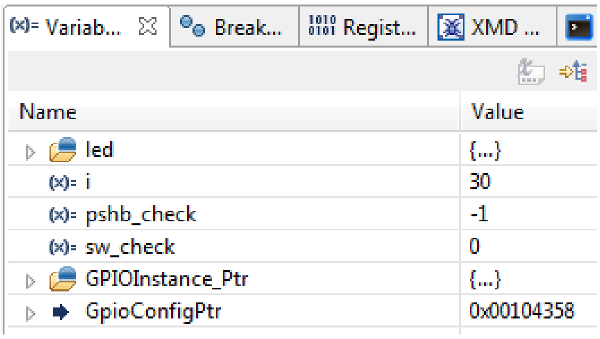
Name	Value
sw	{...}
(x)= BaseAddress	0x41200000
(x)= IsReady	286331153
(x)= InterruptPresent	0
(x)= IsDual	0
led	{...}
(x)= BaseAddress	0x41210000
(x)= IsReady	286331153
(x)= InterruptPresent	0
(x)= IsDual	0

As you realized the base addresses for the `sw` and `led` are different, and they should match the addresses assigned to their respective *GPIO IP* block in the Address Editor in Vivado.



Cell	Slave In...	Base N...	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
board_sw_8b	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
board_leds_8b	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF

2-1-17. At this point the variables *sw_check* and *pshb_check* have any (garbage) value, since they have not been assigned any value yet.



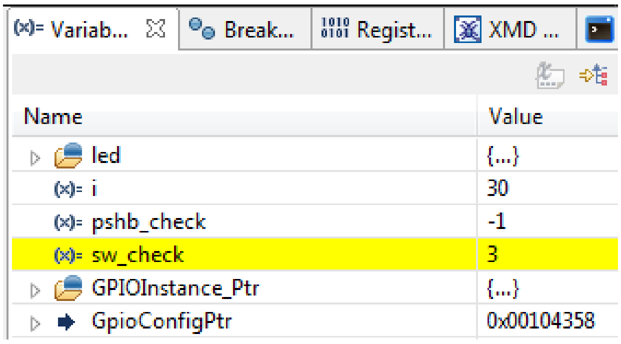
Name	Value
led	{...}
i	30
pshb_check	-1
sw_check	0
GPIOInstance_Ptr	{...}
GpioConfigPtr	0x00104358

2-1-18. Before going to the next step, that will be reading the value of the dip switches on the ZedBoard, let's switch the switches on the board to have the binary value of "0000_0011" (decimal 3).

Note: Don't go to the next step until you move the switches.

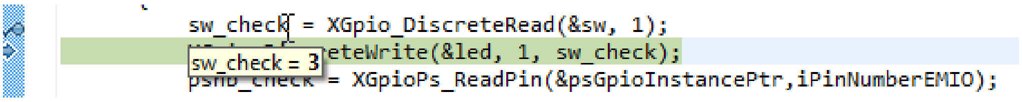
2-1-19. Now, let's click the "*Step Over*" button, to execute the `XGpio_DiscreteRead()` function (the code execution should be halt on this function as it was explained on 2-1-16).

Two main things should have happened in the variable pane: *sw_check* changes to value '3' (in decimal) and *sw_check* is also highlighted in yellow. A variable is highlighted in yellow when the variable has been updated since the last breakpoint or during the last single-step (step-over, step-into) operation. This feature of the debugger can be very useful because it draws your attention to variables that change during the last steps of your debugging session.



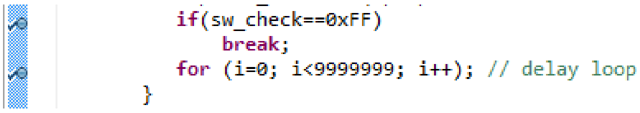
Name	Value
led	{...}
i	30
pshb_check	-1
sw_check	3
GPIOInstance_Ptr	{...}
GpioConfigPtr	0x00104358

The *Variables* pane is not the only place where you can explore the value of variables. In the code editing window, hover your mouse over the name of a variable or a struct. A pop up helper will be displayed and it will give you the same information as in the Variable pane.



```
sw_check = XGpio_DiscreteRead(&sw, 1);
         eteWrite(&led, 1, sw_check);
sw_check = 3
           = XGpioPs_ReadPin(&psGpioInstancePtr,iPinNumberEMIO);
```

2-1-20. Add another breakpoint in the line *if(sw_check==0xFF)* and inside the loop function, as show below.



```
    if(sw_check==0xFF)
        break;
    for (i=0; i<9999999; i++); // delay loop
}
```

2-1-21. If you remember this code is running until we place all the switches to value '1', hence they will be read as "1111_1111" or better, as "0xFF".

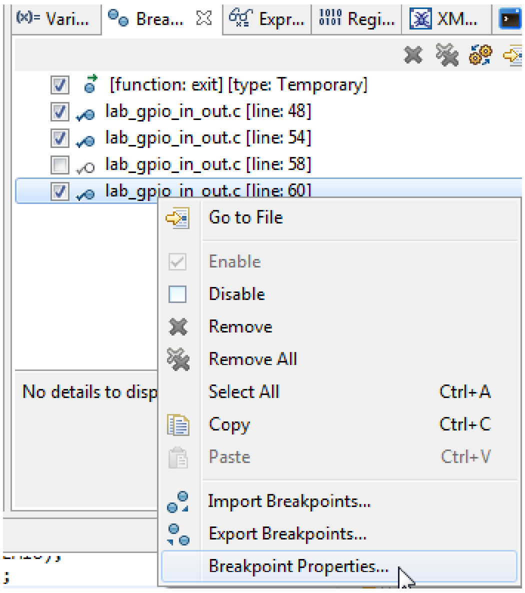
Execute the code until you reach the line `if(sw_check==0xFF)`. At this point the variable `sw_check` can have any value. What we are going to do now is to assign to the variable `sw_check` a specific value. Let's do a double click over the actual value of the `sw_check` variable, then let's write the value 255. Now click the "*Resume*" button and notice that the application code will end since the `if(sw_check==0xFF)` becomes true. By using this feature, we can manually updated the value of any variable to set up a specific condition in the software. This can be used to debug a known set of events, which might be causing a problem in your application.

2-1-22. Breakpoints can be enable and disable: Select the "*Breakpoints*" tab in the top right of the screen, and you will see a list of the breakpoints that are currently set. Experiment with the tick boxes to the left of the breakpoints disabling the break point related to the condition that checks the value of `sw_check`**.**

Conditional Breakpoints

2-1-23. One important application of the breakpoints is to stop the application **only** under certain condition, or for instance, to stop a for-loop in a particular value without the need of doing hundreds of hundreds of step-clicks. This feature is called '*Conditional Breakpoint'*. Hence, a '*Conditional Breakpoint'* is one where the execution of the code will only stop (break point) when a specified condition is met.

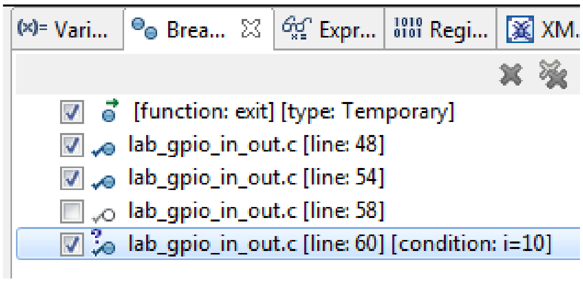
Let's stop the execution of the code when the counter variable of the loop, variable 'i', be equal to '10'. First, you need to select the *Breakpoints* tab, and then select the breakpoint you will work with. Then, right click of the mouse and select '*Breakpoint Properties'*.



☑ [function: exit] [type: Temporary]
☑ lab_gpio_in_out.c [line: 48]
☑ lab_gpio_in_out.c [line: 54]
☐ lab_gpio_in_out.c [line: 58]
☑ lab_gpio_in_out.c [line: 60]

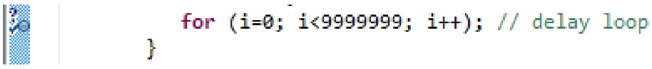
No details to disp

In the '*Properties for C/C++ Line Breakpoint*" window, select '*Common'* and then set the '*Condition'*. In our case is '*i=10*'. Now the *Breakpoint* tab should be updated indicating there is a condition for the breakpoint in line: 60 (you may have another line number).



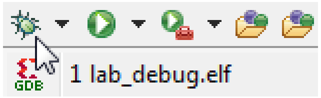
☑ [function: exit] [type: Temporary]
☑ lab_gpio_in_out.c [line: 48]
☑ lab_gpio_in_out.c [line: 54]
□ lab_gpio_in_out.c [line: 58]
☑ lab_gpio_in_out.c [line: 60] [condition: i=10]

In the code editor window there is also an indicator of a conditional breakpoint, the symbol ‘?’, above the usual breakpoint checkmark symbol, now shows up.

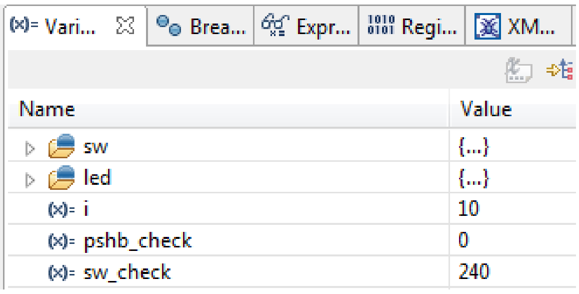


```
        for (i=0; i<9999999; i++); // delay loop
    }
```

2-1-24. Go back to the Variable tab. Click the “Resume” button (if you finished successfully previous steps, likely you need to start the debug again by doing:



2-1-25. Then click ‘*Resume’* and the execution will stop at the enabled breakpoints. Click ‘Resume’ again until the execution stops in the loop statement. Check the value of the variable ‘i’, it should be ‘10’ (be sure you are using other value than “0xFF” in the dip switches).



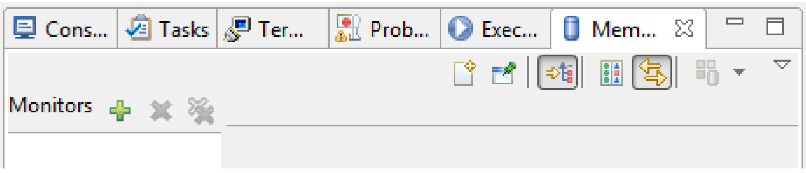
Name	Value
▷ sw	{...}
▷ led	{...}
(x)= i	10
(x)= pshb_check	0
(x)= sw_check	240

2-1-26. Click “*Resume*” again and it will retake the execution of the application.

This part of the lab was an example of a conditional breakpoint that can be very useful for debugging purposes.

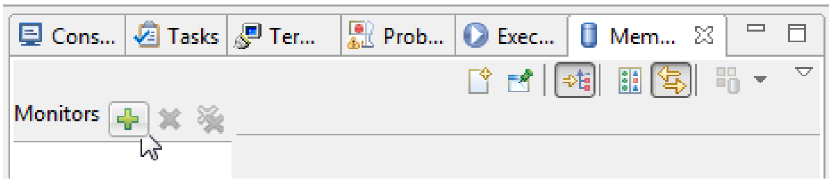
Memory Monitoring

2-1-27. Another tool that is useful for debugging purposes is what is called ‘*Memory Monitoring*’, that is to watch specific memory locations of our interest. In the ‘*Message and Memory View*’ 5 pane, select the memory tab.

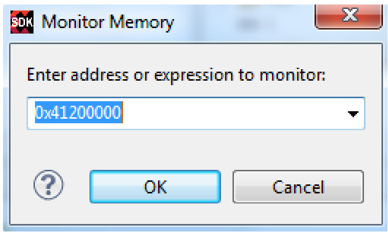


2-1-28. In step 2-1-16 we find out that the memory location associated to `sw_check` is “0x4120_0000”. So, let’s monitoring this memory location to see how it changes when we change the dip switches values.

Click over the “*Add Memory Monitoring’* icon:

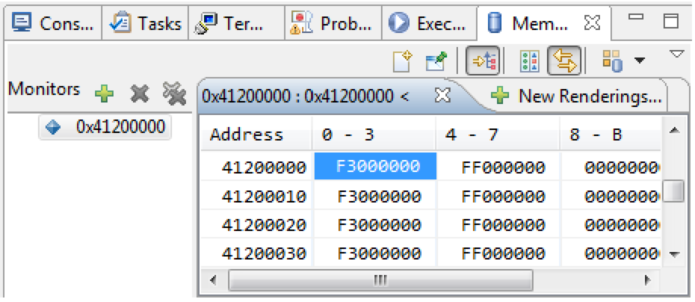


In the Monitor Memory window write down the address to monitor “0x41200000” (to avoid mistyping you can copy the address from the ‘*Variables’* window and past it into here).



Enter address or expression to monitor: 0x41200000

After clicking Ok, the address to monitor along with the neighbor addresses will show up in the memory tab.



Address	0 - 3	4 - 7	8 - B
41200000	F3000000	FF000000	0000000
41200010	F3000000	FF000000	0000000
41200020	F3000000	FF000000	0000000
41200030	F3000000	FF000000	0000000

Remember that the ARM is a 32 bit wide bus, therefore the first 8 bits correspond to the ‘base address’, then the other 8 bits correspond to the ‘base address + 1’, then ‘base address + 2’ and ‘base address + 3’, to complete the 32 bits of the first column of the first row.

2-1-29. Try changing the values of the dip switches, execute the application and the value of the memory address should change accordingly.

Conclusion

Finding the source of an error message is a tough task, knowing debugging skills will improve the way you approach to the possible solution of the problem. This lab presents different tools available in the SDK environment that facilitate the task of finding an error; it does not mean you **will** find the error, but you have learn different approaches that will help you out to find it.